APPENDIX V

SSI_Daemon.cpp

```
// for 2000
#include "stdafx.h"
/********************  EnumProc.h  ********************/
#include <windows.h>


typedef BOOL (CALLBACK *PROCENUMPROC)( DWORD, WORD, LPCSTR, LPARAM );
BOOL WINAPI EnumProcs( PROCENUMPROC lpProc, LPARAM lParam );
//James add
BOOL CALLBACK EnumWindowsProc(HWND hwnd,LPARAM lParam);
//end

/********************  EnumProc.c (or .cpp)  ********************/
//#include "EnumProc.h"
#include <tlhelp32.h>
#include <vdmdbg.h>

#include "stdio.h"

typedef struct
        {
        DWORD       dwPID;
        PROCENUMPROC  lpProc;
        DWORD       lParam;
        BOOL        bEnd;
        } EnumInfoStruct;

// to use this function, declare the following
//BOOL CALLBACK Proc ( DWORD dw, WORD w16, LPCSTR lpstr, LPARAM lParam );


// arrays of start and current processor list
const max_count = 35;
PROCESSENTRY32 startProcs[max_count];
PROCESSENTRY32 currentProcs[max_count];
PROCESSENTRY32 validProcs[max_count];
BOOL firstTime;


BOOL WINAPI Enum16( DWORD dwThreadId, WORD hMod16, WORD hTask16,
    PSZ pszModName, PSZ pszFileName, LPARAM lpUserDefined ) ;
BOOL GetWordPath(LPOLESTR szApp, LPSTR szPath, ULONG cSize);

void nullCurrentProcList();
void killAllNonValidProcs();
BOOL CALLBACK Proc( DWORD PID, WORD w16,LPCSTR lpstr, LPARAM lParam )
{

        LONG *count = (LONG *) lParam;
        if(lpstr !=NULL && strlen(lpstr))
        {
```

```
                if (firstTime == TRUE)
                {
                        startProcs[*count].th32ProcessID = PID;
                        startProcs[*count].cntThreads = 0;
                        strcpy(startProcs[*count].szExeFile, lpstr);
                }
                else
                {
                        currentProcs[*count].th32ProcessID = PID;
                        currentProcs[*count].cntThreads = 0;
                        strcpy(currentProcs[*count].szExeFile, lpstr);
                }
                (*count)++;
        }
        return TRUE;
}


// The EnumProcs function takes a pointer to a callback function
// that will be called once per process in the system providing
// process EXE filename and process ID.
// Callback function definition:
// BOOL CALLBACK Proc( DWORD dw, LPCSTR lpstr, LPARAM lParam );
//
// lpProc -- Address of callback routine.
//
// lParam -- A user-defined LPARAM value to be passed to
//       the callback routine.
BOOL WINAPI EnumProcs( PROCENUMPROC lpProc, LPARAM lParam )
{
        OSVERSIONINFO  osver;
        HINSTANCE    hInstLib;
        HINSTANCE    hInstLib2;
        HANDLE       hSnapShot;
        PROCESSENTRY32 procentry;
        BOOL         bFlag;
        LPDWORD      lpdwPIDs;
        DWORD        dwSize, dwSize2, dwIndex;
        HMODULE      hMod;
        HANDLE       hProcess;
        char         szFileName[ MAX_PATH ];
        EnumInfoStruct sInfo;

        //char display[100];

        // ToolHelp Function Pointers.
        HANDLE (WINAPI *lpfCreateToolhelp32Snapshot)(DWORD,DWORD);
        BOOL (WINAPI *lpfProcess32First)(HANDLE,LPPROCESSENTRY32);
        BOOL (WINAPI *lpfProcess32Next)(HANDLE,LPPROCESSENTRY32);

        // PSAPI Function Pointers.
        BOOL (WINAPI *lpfEnumProcesses)( DWORD *, DWORD cb, DWORD * );
        BOOL (WINAPI *lpfEnumProcessModules)( HANDLE, HMODULE *, DWORD, LPDWORD );
        DWORD (WINAPI *lpfGetModuleFileNameEx)( HANDLE, HMODULE, LPTSTR, DWORD );

        // VDMDBG Function Pointers.
```

```
        INT (WINAPI *lpfVDMEnumTaskWOWEx)( DWORD, TASKENUMPROCEX  fp, LPARAM
);

        // Check to see if were running under Windows95 or Windows NT.
        osver.dwOSVersionInfoSize = sizeof( osver );
        if( !GetVersionEx( &osver ) )
        {
                return FALSE;
        }

        // If Windows NT:
        if( osver.dwPlatformId == VER_PLATFORM_WIN32_NT )
        {
                // Load library and get the procedures explicitly. We do
                // this so that we don't have to worry about modules using
                // this code failing to load under Windows 95, because
                // it can't resolve references to the PSAPI.DLL.
                hInstLib = LoadLibraryA( "PSAPI.DLL" );
                if( hInstLib == NULL )
                        return FALSE;

                hInstLib2 = LoadLibraryA( "VDMDBG.DLL" );
                if( hInstLib2 == NULL )
                        return FALSE ;

                // Get procedure addresses.
                lpfEnumProcesses = (BOOL(WINAPI *)(DWORD *,DWORD,DWORD*))
                        GetProcAddress( hInstLib, "EnumProcesses" );
                lpfEnumProcessModules = (BOOL(WINAPI *)(HANDLE, HMODULE *, DWORD,
LPDWORD))
                        GetProcAddress( hInstLib, "EnumProcessModules" );
                lpfGetModuleFileNameEx =(DWORD (WINAPI *)(HANDLE, HMODULE, LPTSTR,
DWORD ))
                        GetProcAddress( hInstLib, "GetModuleFileNameExA" );
                lpfVDMEnumTaskWOWEx =(INT(WINAPI *)( DWORD, TASKENUMPROCEX,
LPARAM))
                        GetProcAddress( hInstLib2, "VDMEnumTaskWOWEx" );

                if( lpfEnumProcesses == NULL || lpfEnumProcessModules == NULL ||
                        lpfGetModuleFileNameEx == NULL || lpfVDMEnumTaskWOWEx == NULL)
                {
                        FreeLibrary( hInstLib );
                        FreeLibrary( hInstLib2 );
                        return FALSE;
                }

                // Call the PSAPI function EnumProcesses to get all of the
                // ProcID's currently in the system.
                // NOTE: In the documentation, the third parameter of
                // EnumProcesses is named cbNeeded, which implies that you
                // can call the function once to find out how much space to
                // allocate for a buffer and again to fill the buffer.
                // This is not the case. The cbNeeded parameter returns
                // the number of PIDs returned, so if your buffer size is
                // zero cbNeeded returns zero.
                // NOTE: The "HeapAlloc" loop here ensures that we actually
```

```
                // allocate a buffer large enough for all the PIDs in the system.
                dwSize2 = 256 * sizeof( DWORD );
                lpdwPIDs = NULL;
                do
                {
                        if( lpdwPIDs )
        {
                                HeapFree( GetProcessHeap(), 0, lpdwPIDs );
                                dwSize2 *= 2;
                        }
        lpdwPIDs = (LPDWORD)HeapAlloc( GetProcessHeap(), 0, dwSize2 );
        if( lpdwPIDs == NULL )
                        {
                                FreeLibrary( hInstLib );
                                FreeLibrary( hInstLib2 );
                                return FALSE;
                        }
                        if( !lpfEnumProcesses( lpdwPIDs, dwSize2, &dwSize ) )
                        {
                                HeapFree( GetProcessHeap(), 0, lpdwPIDs );
                                FreeLibrary( hInstLib );
                                FreeLibrary( hInstLib2 );
                                return FALSE;
        }
                }
                while( dwSize == dwSize2 );

                // How many ProcID's did we get?
                dwSize /= sizeof( DWORD );

                // Loop through each ProcID.
                for( dwIndex = 0 ; dwIndex < dwSize ; dwIndex++ )
                {
                        szFileName[0] = 0;
                        // Open the process (if we can... security does not
                        // permit every process in the system).
                        hProcess = OpenProcess( PROCESS_QUERY_INFORMATION |
PROCESS_VM_READ,
                                                FALSE, lpdwPIDs[ dwIndex ] );
                        if( hProcess != NULL )
                        {
                                // Here we call EnumProcessModules to get only the
                                // first module in the process this is important,
                                // because this will be the .EXE module for which we
                                // will retrieve the full path name in a second.
                                if( lpfEnumProcessModules( hProcess, &hMod, sizeof( hMod ),
&dwSize2 ) )
                                {
                                        // Get Full pathname:
                                        if( !lpfGetModuleFileNameEx( hProcess, hMod, szFileName,
sizeof( szFileName ) ) )
                                        {
                                                szFileName[0] = 0;
                                        }
                                }
                                CloseHandle( hProcess );
```

```
                        }

                        // Regardless of OpenProcess success or failure, we
                        // still call the enum func with the ProcID.
                        if(!lpProc( lpdwPIDs[dwIndex], 0, szFileName, lParam))
                                break;

                        // Did we just bump into an NTVDM?
                        if( _stricmp( szFileName+(strlen(szFileName)-9), "NTVDM.EXE")==0)
                        {
                                // Fill in some info for the 16-bit enum proc.
                                sInfo.dwPID = lpdwPIDs[dwIndex];
                                sInfo.lpProc = lpProc;
                                sInfo.lParam = lParam;
                                sInfo.bEnd = FALSE;
                                // Enum the 16-bit stuff.
                                lpfVDMEnumTaskWOWEx( lpdwPIDs[dwIndex],
(TASKENUMPROCEX) Enum16,
                                        (LPARAM) &sInfo);
                                // Did our main enum func say quit?
                                if(sInfo.bEnd)
                                        break;
                        }
                }
                HeapFree( GetProcessHeap(), 0, lpdwPIDs );
                FreeLibrary( hInstLib2 );

        // If Windows 95:
        }
        else if( osver.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS )
        {
                hInstLib = LoadLibraryA( "Kernel32.DLL" );
                if( hInstLib == NULL )
                        return FALSE;

                // Get procedure addresses.
                // We are linking to these functions of Kernel32 explicitly, because
                // otherwise a module using this code would fail to load under Windows NT,
                // which does not have the Toolhelp32 functions in the Kernel 32.
                lpfCreateToolhelp32Snapshot= (HANDLE(WINAPI *)(DWORD,DWORD))
                        GetProcAddress( hInstLib, "CreateToolhelp32Snapshot" );
                lpfProcess32First= (BOOL(WINAPI *)(HANDLE,LPPROCESSENTRY32))
                        GetProcAddress( hInstLib, "Process32First" );
                lpfProcess32Next= (BOOL(WINAPI *)(HANDLE,LPPROCESSENTRY32))
                        GetProcAddress( hInstLib, "Process32Next" );

                if( lpfProcess32Next == NULL || lpfProcess32First == NULL ||
                        lpfCreateToolhelp32Snapshot == NULL )
                {
                        FreeLibrary( hInstLib );
                        return FALSE;
                }

                // Get a handle to a Toolhelp snapshot of the systems processes.
                hSnapShot = lpfCreateToolhelp32Snapshot( TH32CS_SNAPPROCESS, 0 );
```

```
		if( hSnapShot == INVALID_HANDLE_VALUE )
		{
			FreeLibrary( hInstLib );
	return FALSE;
		}

		// Get the first process' information.
		procentry.dwSize = sizeof(PROCESSENTRY32);
		bFlag = lpfProcess32First( hSnapShot, &procentry );

		while( bFlag )
		{
			//itoa(procentry.th32ProcessID, display, 16);
			//MessageBox( NULL, display, "Proc Killer 95 and NT", MB_OK );
			// Call the enum func with the filename and ProcID.
			if(lpProc( procentry.th32ProcessID, 0, procentry.szExeFile, lParam ))
			{
				procentry.dwSize = sizeof(PROCESSENTRY32);
				bFlag = lpfProcess32Next( hSnapShot, &procentry );
			}
			else
				bFlag = FALSE;
		}
		CloseHandle(hSnapShot);
	}
	else
		return FALSE;

	if (firstTime == TRUE)
		firstTime = FALSE;

	// Free the library.
	FreeLibrary( hInstLib );

	return TRUE;
}


BOOL WINAPI Enum16( DWORD dwThreadId, WORD hMod16, WORD hTask16,
	PSZ pszModName, PSZ pszFileName, LPARAM lpUserDefined )
{
	BOOL bRet;
	EnumInfoStruct *psInfo = (EnumInfoStruct *)lpUserDefined;
	bRet = psInfo->lpProc( psInfo->dwPID, hTask16, pszFileName, psInfo->lParam );

	if(!bRet)
	{
		psInfo->bEnd = TRUE;
	}

	return !bRet;
}

////////////////////////////////
// null out the current proc list
```

```
////////////////////////////////
void nullCurrentProcList()
{
        for (int i = 0; i < max_count; i++)
        {
                currentProcs[i].th32ProcessID = 0;
                currentProcs[i].cntThreads = 0;
                strcpy(currentProcs[i].szExeFile, "");
        }
}

////////////////////////////////
// kill all non valid procs
////////////////////////////////

void killAllNonValidProcs()
{
        PROCENUMPROC lpProc;
        LONG  lParam;
        HANDLE procToKill;
        DWORD dwDesiredAccess;
        BOOL bInheritHandle;
        DWORD dwProcessId;
        FILE *fp_pids;              // PIDs file
        FILE *fp_torestart;         // file of processes that must be restarted all killed procs)
        int termVal; // is 0 if the process does not terminate
        char lpszRetStr[255];

        nullCurrentProcList();
        lParam=0;
        lpProc= Proc;
        EnumProcs( lpProc, (LPARAM) &lParam );

        // this will empty the restart file if it is not already null
        fp_torestart = fopen("c:\\restartpids.bat", "w");
        fclose(fp_torestart);

        fp_pids = fopen("c:\\pids.txt", "a");
        fprintf(fp_pids, "\n\nSearching Procs to kill:\n");
        fprintf(fp_pids,    "==========================\n");
        fclose(fp_pids);



        char szApp[80];
        LPOLESTR szwApp;
        strcpy(szApp,"Word.Application");
                // Find number ofcharacters to be allocated
        int len2 = strlen(szApp) + 1;

                // Use OLE Allocator to allocate memory
                szwApp = (LPOLESTR) CoTaskMemAlloc(len2*2);
                if (szwApp == NULL)
                {
//              MessageBox("Out of Memory", "Error");
                return ;
```

```
                }

                //          AnsiToUnicode conversion
                if (0 == MultiByteToWideChar(CP_ACP, 0, szApp, len2,
                                szwApp, len2))
                {
                  // Free Memory allocated to szwApp if conversion failed
                  CoTaskMemFree(szwApp);
                  szwApp = NULL;
        //      MessageBox("Error in Conversion", "Error");
                  return ;
                }

                // Get Path to Application and display it
                GetWordPath(szwApp, lpszRetStr, 255);
        char szSysPath[255];
        long len;
        long lenWinDir;
        GetSystemDirectory(szSysPath,sizeof(szSysPath));
        len = strlen(szSysPath);
        // kill all non-essential procs
        char szWinDir[255];
        GetWindowsDirectory(szWinDir,sizeof(szWinDir));
        lenWinDir = strlen(szWinDir);
        char szTaskMon[255];
        strcpy(szTaskMon,szWinDir);
        strcat(szTaskMon,"\\TASKMON.EXE");
        for (int i = 0; i < max_count; i++)
        {
                char szShortPath[255];
                GetShortPathName(currentProcs[i].szExeFile,szShortPath,255);
                if (stricmp(&(currentProcs[i].szExeFile[len+1]),"KERNEL32.DLL") == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"MSGSRV32.EXE") == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"INTERNAT.EXE") == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"MPREXE.EXE") == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"MSTASK.EXE") == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"RUNONCE.EXE") == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"RPCSS.EXE") == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"SPOOL32.EXE") == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"SSI_TIMER.DLL") == 0 ||
//                      stricmp(&(currentProcs[i].szExeFile[lenWinDir+1]),"EXPLORER.EXE") == 0
||

                        stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\SECUREXAM
STUDENT\\SSI_STUDENT.EXE") == 0 ||

        stricmp(currentProcs[i].szExeFile,"C:\\WINDOWS\\DESKTOP\\SSI_STUDENT.EXE") == 0 ||
//                      strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\MICROSOFT
OFFICE\\OFFICE\\WINWORD.EXE") == 0 ||
                        stricmp(szShortPath,lpszRetStr) == 0 ||
                        //strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM
FILES\\WEBSVR\\SYSTEM\\INETSW95.EXE") == 0 ||
                        //strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\NORTON
ANTIVIRUS\\NAVAPW32.EXE") == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"mmtask.tsk") == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"PSTORES.EXE") == 0 ||
```

```
                        strcmp(currentProcs[i].szExeFile,szTaskMon) == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"SYSTRAY.EXE") == 0 ||
                        //strcmp(currentProcs[i].szExeFile,"C:\\WINDOWS\\ESSOLO.EXE") == 0 ||
                        stricmp(currentProcs[i].szExeFile,"C:\\MOUSE\\SYSTEM\\EM_EXEC.EXE")
== 0 ||

        //strcmp(currentProcs[i].szExeFile,"C:\\IBMTOOLS\\APTEZBTN\\APTEZBP.EXE") == 0 ||
                        //strcmp(currentProcs[i].szExeFile,"C:\\CSAFE\\AUTOCHK.EXE") == 0 ||
                        //strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM
FILES\\REAL\\REALPLAYER\\REALPLAY.EXE") == 0 ||
                        //strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\ICQ\\ICQ.EXE")
== 0 ||
                        //strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\NORTON
ANTIVIRUS\\NSCHED32.EXE") == 0 ||
                        //strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\MICROSOFT
OFFICE\\OFFICE\\OSA.EXE") == 0 ||
        //              strcmp(currentProcs[i].szExeFile,"C:\\TOOLS_95\\IOWATCH.EXE") == 0 ||
        //              strcmp(currentProcs[i].szExeFile,"C:\\TOOLS_95\\IMGICON.EXE") == 0 ||
        //              strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM
FILES\\DEVSTUDIO\\SHAREDIDE\\BIN\\MSDEV.EXE") == 0 ||
//                      stricmp(&(currentProcs[i].szExeFile[len+1]),"WINOA386.MOD") == 0 ||


//----jadder ------old
//                      stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\SECUREXAM
STUDENT\\STOP_SSI_DAEMON.EXE") == 0 ||
                        stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\SECUREXAM
STUDENT\\SSI_DAEMON.EXE") == 0 ||
//---j Rep
                        stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\SECUREXAM
STUDENT\\SSI_Temp.dat") == 0 ||   //  <--othee file
                        stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\SECUREXAM
STUDENT\\SSITmpST.dat") == 0 ||   //  <--stop_ssi_daemon
                        stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\SECUREXAM
STUDENT\\SSITemp2.dat") == 0 )    ///  <--ssi_daemon
//----j end

                {
                        // do nothing, these are ok
                }
                else
                {
                        dwProcessId = currentProcs[i].th32ProcessID;
                        if (dwProcessId != 0)
                        {
                                // kill these
                                dwDesiredAccess = PROCESS_ALL_ACCESS;
                                bInheritHandle =TRUE;
                                procToKill = OpenProcess( dwDesiredAccess, bInheritHandle,
dwProcessId );
                                termVal = TerminateProcess(procToKill, 0);

// Who          : Robin wei
// Date         : 02-9-24 14:52:53
// Reason       : To make sure the process has been terminated and clear the object.
// Modify ---------------------------------- [Begin]
```

```
//                                              WaitForSingleObject(procToKill,INFINITE);
                                                CloseHandle(procToKill);
// Modify ------------------------------------ [End]

                                        if (termVal != 0)
                                        {
                                                fp_pids = fopen("c:\\pids.txt", "a");
                                                fprintf(fp_pids, "Proc KILLED: 0x%x  %s\n",
currentProcs[i].th32ProcessID, currentProcs[i].szExeFile);
                                                fclose(fp_pids);

                                                // save the procs that must be restarted at end of exam to a .bat
file
                                                fp_torestart = fopen("c:\\restartpids.bat", "a+");
                                                fprintf(fp_torestart, "\"%s\"\n", currentProcs[i].szExeFile);
                                                fclose(fp_torestart);
                                        }
                                }
                        }
                }
                // append synchronization file creation to the end of the restart .bat file
                fp_torestart = fopen("c:\\restartpids.bat", "a+");
                fprintf(fp_torestart, "echo \"RESTART SYNC FILE\" > c:\\dumdumresfile.txt" );
                fclose(fp_torestart);
}


int main(int argc, char* argv[])
{
        PROCENUMPROC lpProc;
        LPARAM lParam;
        HANDLE procToKill;
        DWORD dwDesiredAccess;
        BOOL bInheritHandle, procIsOK;
        DWORD dwProcessId;
        FILE *fp_pids;   // PIDs file
        FILE *fp_cheat;  // cheat file
        int i, j, num_valid;

        // kick off the SSI_STUDENT.exe
        //system( "c:\\tom\\procKiller95andNT\\SSI_STUDENT.exe" );
        CoInitialize(NULL);

        // init the start, current, and valid proc lists
        for (i = 0; i < max_count; i++)
        {
                startProcs[i].th32ProcessID = 0;
                startProcs[i].cntThreads = 0;
                strcpy(startProcs[i].szExeFile, "");

                validProcs[i].th32ProcessID = 0;
                validProcs[i].cntThreads = 0;
                strcpy(validProcs[i].szExeFile, "");
        }

        nullCurrentProcList(); // clear the current proc list
```

```
        // get snapshot of starting processes
        firstTime = TRUE;
        lParam=0;
        lpProc= Proc;
        EnumProcs( lpProc, (LPARAM) (&lParam) );
        firstTime = FALSE;

        // write out starting processes to file
        fp_pids = fopen("c:\\pids.txt", "w+");
        fprintf(fp_pids, "Starting PIDS:\n");
        fprintf(fp_pids, "==============\n");
        for (i = 0; i < max_count; i++)
        {
                if (startProcs[i].th32ProcessID != 0)
                {
                        fprintf(fp_pids, "0x%x %ld %s\n", startProcs[i].th32ProcessID,
                        startProcs[i].cntThreads, startProcs[i].szExeFile);
                }
        }
        fclose(fp_pids);

        // delete all non-essential processes
        killAllNonValidProcs();
FreeConsole();
        Sleep(500);  //allow settling

        // take a snapshot - the remaining procs are valid
        nullCurrentProcList(); // clear the current proc list
        lParam=0;
        lpProc= Proc;
        EnumProcs( lpProc, (LPARAM) &lParam );


        // The current proc list now has all of the valid procs allowed.
        // Copy these to the valid proc list to be used for cheat detection.
        fp_pids = fopen("c:\\pids.txt", "a");
        fprintf(fp_pids, "\n\nValid Procs:\n");
        fprintf(fp_pids,    "=============\n");
        num_valid = 0;
        for (i = 0; i < max_count; i++)
        {
                if (currentProcs[i].th32ProcessID != 0)
                {
                        validProcs[i].th32ProcessID = currentProcs[i].th32ProcessID;
                        validProcs[i].cntThreads = currentProcs[i].cntThreads;
                        strcpy(validProcs[i].szExeFile, currentProcs[i].szExeFile);

                        fprintf(fp_pids, "0x%x %ld %s\n", validProcs[i].th32ProcessID,
                        validProcs[i].cntThreads, validProcs[i].szExeFile);
                        num_valid++;
                }
        }
        fclose(fp_pids);
        char szSysPath[255];
        long len;
```

```
GetSystemDirectory(szSysPath,sizeof(szSysPath));
len = strlen(szSysPath);

// main cheat detection, kill opening procs and record to cheat file
while(1)
{
	// Main message loop:
	MSG msg;
	while (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
	{
		TranslateMessage(&msg);
		DispatchMessage(&msg);
	}
	Sleep(500);//tpr 5/1/99

	nullCurrentProcList(); // clear the current proc list
	lParam=0;
	lpProc= Proc;
	EnumProcs( lpProc, (LPARAM) &lParam );


	HWND hwnd;
	hwnd = FindWindow("WinPopup",NULL);
	if(hwnd )
	{
		Sleep(1000);
		continue;
	}
	hwnd = FindWindow("ExploreWClass",NULL);
	if(hwnd)
	{
		PostMessage(hwnd,WM_CLOSE,0,0);
	}
	hwnd = FindWindow("CabinetWClass",NULL);
	if(hwnd)
	{
		PostMessage(hwnd,WM_CLOSE,0,0);
	}
	for (i=0; i < max_count; i++)
	{
		if (currentProcs[i].th32ProcessID != 0)
		{
			procIsOK = FALSE;
			for (j=0; j < num_valid; j++)
			{
				if (currentProcs[i].th32ProcessID ==
validProcs[j].th32ProcessID)
					procIsOK = TRUE;
			}

//---jadder---old
//				if(	stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM
FILES\\SECUREXAM STUDENT\\STOP_SSI_DAEMON.EXE") == 0 ||
// ---j Rep
				if(	stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM
FILES\\SECUREXAM STUDENT\\SSI_Temp.dat") == 0 ||
```

```
//---j end

                                stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM
FILES\\SECUREXAM STUDENT\\SSITmpST.dat") == 0 ||

//
        stricmp(&(currentProcs[i].szExeFile[len+1]),"WINOA386.MOD") == 0 ||

        stricmp(&(currentProcs[i].szExeFile[len+1]),"SSI_TIMER.DLL") == 0 )
                                {
                                        procIsOK = TRUE;
                                }

                                if (procIsOK == FALSE)
                                {
                                        dwDesiredAccess = PROCESS_ALL_ACCESS;
                                        bInheritHandle =TRUE;
                                        dwProcessId = currentProcs[i].th32ProcessID;
                                        procToKill = OpenProcess( dwDesiredAccess,
bInheritHandle, dwProcessId );
                                        TerminateProcess(procToKill, 0);
// Who          : Robin wei
// Date         : 02-9-24 14:52:53
// Reason       : To make sure the process has been terminated and clear the object.
// Modify ---------------------------------- [Begin]
//                                      WaitForSingleObject(procToKill,INFINITE);
                                        CloseHandle(procToKill);
// Modify ---------------------------------- [End]

                                        //fp_pids = fopen("c:\\pids.txt", "a");
                                        //fprintf(fp_pids, "\nCHEATING DETECTED using
%s\n",currentProcs[i].szExeFile);
                                        //fclose(fp_pids);
                                        fp_cheat = fopen("c:\\cheatfile.txt", "a");
                                        fprintf(fp_cheat, "\nCHEATING DETECTED using
%s\n",currentProcs[i].szExeFile);
                                        fclose(fp_cheat);

                                        //exit(1); // must be killed by the SSI Student app.
                                }
                        }
                }

//James add
//----------------------------begin

//              EnumWindows(EnumWindowsProc,0);
//              EnumDesktopWindows(NULL,EnumWindowsProc,0);
//----------------------------end



        }//while
        return 0;
}
```

```
BOOL GetWordPath(LPOLESTR szApp, LPSTR szPath, ULONG cSize)
{
  CLSID clsid;
  LPOLESTR pwszClsid;
  CHAR  szKey[128];
  CHAR  szCLSID[60];
  HKEY hKey;
        ULONG oldSize = cSize;
  // szPath must be at least 255 char in size
  if (cSize < 255)
        return FALSE;

  // Get the CLSID using ProgID
  HRESULT hr = CLSIDFromProgID(szApp, &clsid);
  if (FAILED(hr))
  {
//        AfxMessageBox("Could not get CLSID from ProgID, Make sure ProgID is correct", MB_OK, 0);
    return FALSE;
  }

  // Convert CLSID to String
  hr = StringFromCLSID(clsid, &pwszClsid);
  if (FAILED(hr))
  {
//        AfxMessageBox("Could not convert CLSID to String", MB_OK, 0);
    return FALSE;
  }

  // Convert result to ANSI
  WideCharToMultiByte(CP_ACP, 0, pwszClsid, -1, szCLSID, 60, NULL, NULL);

  // Free memory used by StringFromCLSID
  CoTaskMemFree(pwszClsid);

  // Format Registry Key string
  wsprintf(szKey, "CLSID\\%s\\LocalServer32", szCLSID);

  // Open key to find path of application
  LONG lRet = RegOpenKeyEx(HKEY_CLASSES_ROOT, szKey, 0, KEY_ALL_ACCESS, &hKey);
  if (lRet != ERROR_SUCCESS)
  {
        // If LocalServer32 does not work, try with LocalServer
    wsprintf(szKey, "CLSID\\%s\\LocalServer", szCLSID);
    lRet = RegOpenKeyEx(HKEY_CLASSES_ROOT, szKey, 0, KEY_ALL_ACCESS, &hKey);
        if (lRet != ERROR_SUCCESS)
    {
//        AfxMessageBox("No LocalServer Key found!!", MB_OK, 0);
      return FALSE;
    }
  }

  // Query value of key to get Path and close the key
  lRet = RegQueryValueEx(hKey, NULL, NULL, NULL, (BYTE*)szPath, &cSize);
  RegCloseKey(hKey);
  if (lRet != ERROR_SUCCESS)
  {
```

```
//        AfxMessageBox("Error trying to query for path", MB_OK, 0);
    return FALSE;
  }

  // Strip off the '/Automation' switch from the path
  char *x = strrchr(szPath, '/');
  if(0!= x) // If no /Automation switch on the path
  {
  int result = x - szPath;
  szPath[result]  = '\0';  // If switch there, strip it
  }
        for(int i= strlen(szPath)-1;i>=0;i--)
        {
                if(szPath[i] == ' ')
                        szPath[i] =0;
                else
                        break;
        }

// Who              : Robin wei
// Date             : 00-10-8 13:45:03
// Reason           : For compile with Win95
// Modify ---------------------------------- [Begin]
          GetShortPathName(szPath,szPath,oldSize);
// Modify ---------------------------------- [End]


// Who              : Robin wei
// Date             : 00-10-8 13:44:41
// Reason           : This funciton does not exists in win 95
#if 0 // Delete ---------------------------------- [Begin]

                GetLongPathName(szPath,szPath,oldSize);

#endif  // Delete ---------------------------------- [End]

  return TRUE;
}


//James add
BOOL CALLBACK EnumWindowsProc(HWND hwnd,LPARAM lParam)
{
        bool bInvalid;
        DWORD ProID;
        LPDWORD lpdwProcessId=&ProID;
        WINDOWPLACEMENT wndpl;
  GetWindowPlacement(hwnd,&wndpl);
        if (wndpl.showCmd=SW_HIDE)
        {

        GetWindowThreadProcessId(hwnd,lpdwProcessId);
        bInvalid=false;
  for (int i=0; i < max_count; i++)
        {
                if (currentProcs[i].th32ProcessID == *lpdwProcessId)
```

```
                {
                        bInvalid=true;
                        break;
                }
        }
        if (bInvalid==false)
        {
                PostMessage(hwnd,WM_CLOSE,0,0);
        }
        }
        return true;
}

//end
```

stdafx.cpp

```
// stdafx.cpp : source file that includes just the standard includes
//        ssi_daemon.pch will be the pre-compiled header
//        stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"

// TODO: reference any additional headers you need in STDAFX.H
// and not in this file
```

Stdafx.h

```
// stdafx.h : include file for standard system include files,
//  or project specific include files that are used frequently, but
//      are changed infrequently
//

#if !defined(AFX_STDAFX_H__A9DB83DB_A9FD_11D0_BFD1_444553540000__INCLUDED_)
#define AFX_STDAFX_H__A9DB83DB_A9FD_11D0_BFD1_444553540000__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define WIN32_LEAN_AND_MEAN             // Exclude rarely-used stuff from Windows headers

#include <windows.h>
#include <objbase.h>

// TODO: reference additional headers your program requires here

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_STDAFX_H__A9DB83DB_A9FD_11D0_BFD1_444553540000__INCLUDED_)
```

APPENDIX IV

Tempdoc.doc
Module1 (Code)

```
Option Explicit
'
' Listing 25.10. Using Win32 API functions to read the Windows 95 sub version from the Registry.
'
Declare Function RegOpenKeyEx Lib "advapi32.dll" Alias "RegOpenKeyExA" (ByVal hKey As Long,
ByVal lpSubKey As String, ByVal ulOptions As Long, ByVal samDesired As Long, phkResult As Long)
As Long
Declare Function RegQueryValueEx Lib "advapi32.dll" Alias "RegQueryValueExA" (ByVal hKey As
Long, ByVal lpValueName As String, ByVal lpReserved As Long, lpType As Long, ByVal lpData As
Any, lpcbData As Long) As Long
Declare Function RegCloseKey Lib "advapi32.dll" (ByVal hKey As Long) As Long
Declare Function RegFlushKey Lib "advapi32.dll" (ByVal hKey As Long) As Long
Declare Function RegSetValueEx Lib "advapi32.dll" Alias "RegSetValueExA" (ByVal hKey As Long,
ByVal lpValueName As String, ByVal Reserved As Long, ByVal dwType As Long, ByVal lpData As
Any, ByVal cbData As Long) As Long

'
' You may not need all these variables!
'
Public Const DELETE = &H10000
Public Const READ_CONTROL = &H20000
Public Const WRITE_DAC = &H40000
Public Const WRITE_OWNER = &H80000
Public Const SYNCHRONIZE = &H100000
Public Const STANDARD_RIGHTS_READ = (READ_CONTROL)
Public Const STANDARD_RIGHTS_WRITE = (READ_CONTROL)
Public Const STANDARD_RIGHTS_EXECUTE = (READ_CONTROL)
Public Const STANDARD_RIGHTS_REQUIRED = &HF0000
Public Const STANDARD_RIGHTS_ALL = &H1F0000
Public Const KEY_QUERY_VALUE = &H1
Public Const KEY_SET_VALUE = &H2
Public Const KEY_CREATE_SUB_KEY = &H4
Public Const KEY_ENUMERATE_SUB_KEYS = &H8
Public Const KEY_NOTIFY = &H10
Public Const KEY_CREATE_LINK = &H20
Public Const KEY_READ = ((STANDARD_RIGHTS_READ Or KEY_QUERY_VALUE Or
KEY_ENUMERATE_SUB_KEYS Or KEY_NOTIFY) And (Not SYNCHRONIZE))
Public Const KEY_WRITE = ((STANDARD_RIGHTS_WRITE Or KEY_SET_VALUE Or
KEY_CREATE_SUB_KEY) And (Not SYNCHRONIZE))
Public Const KEY_EXECUTE = (KEY_READ)
Public Const KEY_ALL_ACCESS = ((STANDARD_RIGHTS_ALL Or KEY_QUERY_VALUE Or
KEY_SET_VALUE Or KEY_CREATE_SUB_KEY Or KEY_ENUMERATE_SUB_KEYS Or
KEY_NOTIFY Or KEY_CREATE_LINK) And (Not SYNCHRONIZE))
Public Const ERROR_SUCCESS = 0&
Public Const HKEY_CLASSES_ROOT = &H80000000
Public Const HKEY_CURRENT_USER = &H80000001
Public Const HKEY_LOCAL_MACHINE = &H80000002
Public Const HKEY_USERS = &H80000003
Public Const REG_SZ = 1
```

```
-

Public Const REG_BINARY = 3
Public Const REG_DWORD = 4
Dim starttime As Date
Dim oldCustomDic As String
Const wdFieldDocProperty = 85
' ================================================================================
' AutoNew
'    - Fires when a new document is created from the template. Opening
'      the template for editing won't effect you.
'
' ================================================================================
' Sub AutoExec()
' Sub AutoNew()
' Sub AutoOpen() Modified by GAT Bernard Young 30-08-00
Sub MInitialize()

   Dim i As Integer
   On Error Resume Next

   " <--- Added by GAT Bernard Young 30-08-00
   Selection.LanguageID = wdEnglishUS
   Selection.NoProofing = False
   Application.CheckLanguage = False
   Application.EnableCancelKey = wdCancelDisabled
   Application.ActiveWindow.DisplayHorizontalScrollBar = False
   oldCustomDic = Application.CustomDictionaries.ActiveCustomDictionary.Path & "\"
   oldCustomDic = oldCustomDic & Application.CustomDictionaries.ActiveCustomDictionary.Name
   Application.CustomDictionaries.ClearAll
   If (ActiveDocument.CustomDocumentProperties.Count <= 0) Then
      Call ActiveDocument.CustomDocumentProperties.Add("CustomDic", False, msoPropertyTypeString,
oldCustomDic)
   End If

   " ActiveWindow.View.FullScreen = True
   " Ended here ---->
   Dim intCount As Integer
      'To Disable all the Tool Bars ---------
      For intCount = 1 To CommandBars.Count
'        CommandBars(intCount).Visible = False
         CommandBars(intCount).Enabled = False
         If (CommandBars(intCount).Name = "SSI Tool Bar") Then
            CommandBars(intCount).DELETE
         End If
      Next


'
' Macro1 Macro
' Macro recorded 05/02/99 by douglas
'
      CommandBars("menu bar").Enabled = True
      CommandBars("menu bar").Visible = True

      CommandBars("Menu Bar").Position = msoBarTop
   'To Remove all the Menu Items of Microsoft Word
      For i = 1 To 10
```

```
    CommandBars("Menu Bar").Controls(1).DELETE
  Next

  For i = 1 To 10
    CommandBars("Menu Bar").Controls(2).DELETE
  Next

  For i = 1 To 10
    CommandBars("Menu Bar").Controls(3).DELETE
  Next

'Add the SSI Menu
Dim oToolbar As CommandBar
Dim oNewFileMenu As CommandBarPopup

Set oToolbar = CommandBars("Menu Bar")
Set oNewFileMenu = oToolbar.Controls.Add(msoControlPopup, , , , True)
Let oNewFileMenu.Caption = "&File"
Let oNewFileMenu.TooltipText = "File"
Let oNewFileMenu.Visible = True
Dim oNewEditMenu As CommandBarPopup
' Set oToolbar = CommandBars("Menu Bar")
Set oNewEditMenu = oToolbar.Controls.Add(msoControlPopup, , , , True)
Let oNewEditMenu.Caption = "&Edit"
Let oNewEditMenu.TooltipText = "Edit"
Let oNewEditMenu.Visible = True
Dim oNewViewMenu As CommandBarPopup
' Set oToolbar = CommandBars("Menu Bar")
Set oNewViewMenu = oToolbar.Controls.Add(msoControlPopup, , , , True)
Let oNewViewMenu.Caption = "&View"
Let oNewViewMenu.TooltipText = "View"
Let oNewViewMenu.Visible = True
Dim oNewToolsMenu As CommandBarPopup
' Set oToolbar = CommandBars("Menu Bar")
Set oNewToolsMenu = oToolbar.Controls.Add(msoControlPopup, , , , True)
Let oNewToolsMenu.Caption = "&Tools"
Let oNewToolsMenu.TooltipText = "Tools"
Let oNewToolsMenu.Visible = True


' Set oToolbar = CommandBars("Menu Bar")
' Dim oNewTimerMenu As CommandBarButton
' Set oNewTimerMenu = oToolbar.Controls.Add(msoControlButton, , , , True)
' Let oNewTimerMenu.Caption = "Ti&mer"
' With oNewTimerMenu
'   .Style = msoButtonCaption
'   .TooltipText = "Exam Time"
'   .Visible = True
'   .OnAction = "ShowTime"
'   " Added by GAT Bernard Young 05-09-00
'   .ShortcutText = "CTRL + T"
' End With
' oNewTimerMenu.ShortcutText = "ctrl+t"
' Let oNewTimerMenu.Visible = True

  " Added by GAT Bernard Young 09-14-00
```

```
" to make it possible to show exam time
" in Word2000 when security set to high
" by using AddIn


With Application
   For intCount = 1 To .COMAddIns.Count
      If .COMAddIns(intCount).ProgID = "SSI_ShowTime.dsrSSI_Timer" Then
         .COMAddIns(intCount).Connect = True
         Exit For
      End If
   Next
End With

Dim oNewTestMenu As CommandBarPopup
' Set oToolbar = CommandBars("Menu Bar")
Set oNewTestMenu = oToolbar.Controls.Add(msoControlPopup, , , , True)
Let oNewTestMenu.Caption = "To E&xit"
Let oNewTestMenu.TooltipText = "Exit"
Let oNewTestMenu.Visible = True

'Set oToolbar = CommandBars("Menu Bar")
'Set oNewHelpMenu = oToolbar.Controls.Add(msoControlPopup, , , , True)
'Let oNewHelpMenu.Caption = "&Help"
'Let oNewHelpMenu.TooltipText = "Help"
'Let oNewHelpMenu.Visible = True

Call oNewFileMenu.CommandBar.Controls.Add(msoControlButton, 3)

 Call oNewViewMenu.CommandBar.Controls.Add(msoControlButton, 224)
' Call oNewViewMenu.CommandBar.Controls.Add(msoControlButton, 760)
 Call oNewViewMenu.CommandBar.Controls.Add(msoControlButton, 287)


Call oNewEditMenu.CommandBar.Controls.Add(msoControlButton, 21)
Call oNewEditMenu.CommandBar.Controls.Add(msoControlButton, 19)
Call oNewEditMenu.CommandBar.Controls.Add(msoControlButton, 22)
Call oNewEditMenu.CommandBar.Controls.Add(msoControlButton, 141)
Dim oFindNextBar As CommandBarButton
Set oFindNextBar = oNewEditMenu.CommandBar.Controls.Add(msoControlButton, 570)
' oFindNextBar.OnAction = "Selection.Find.Execute"
' oFindNextBar.ShortcutText = "Ctrl+Alt+Y"
Set oFindNextBar = Nothing
Call oNewEditMenu.CommandBar.Controls.Add(msoControlButton, 313)

Call oNewToolsMenu.CommandBar.Controls.Add(msoControlButton, 2)
Call oNewToolsMenu.CommandBar.Controls.Add(msoControlButton, 792)

'Call oNewHelpMenu.CommandBar.Controls.Add(msoControlButton, 983)
'Call oNewHelpMenu.CommandBar.Controls.Add(msoControlButton, 927)

Call oNewTestMenu.CommandBar.Controls.Add(msoControlButton, 752)

Set oNewFileMenu = Nothing
Set oNewEditMenu = Nothing
Set oNewToolsMenu = Nothing
```

```
'Set oNewHelpMenu = Nothing
Set oNewTestMenu = Nothing
Set oNewViewMenu = Nothing

 oToolbar.Protection = msoBarNoChangeDock + msoBarNoChangeVisible + msoBarNoMove
Set oToolbar = Nothing

    With Options
      .SaveNormalPrompt = False
    End With


    Selection.Font.Size = 12
    Selection.Font.Name = "Times New Roman"

    " Added by GAT Bernard Young 31-08-00
    On Error Resume Next
    Assistant.Visible = False
    ActiveWindow.ActivePane.DisplayRulers = True
    CommandBars("Menu Bar").Visible = True
    CommandBars("Menu Bar").Position = msoBarTop
    CommandBars("Full Screen").Visible = False

    Dim ctrlFormatting As CommandBarControl
    Set oToolbar = CommandBars.Add("SSI Tool Bar", msoBarTop, , True)
    For Each ctrlFormatting In CommandBars("Formatting").Controls
      If (ctrlFormatting.ID <> 1732 And ctrlFormatting.BuiltIn = True) Then
        Call ctrlFormatting.Copy(oToolbar)
      End If
""        Call oToolbar.Controls.Add(ctrlFormatting.Type, ctrlFormatting.ID, ctrlFormatting.Parameter, ,
True)
    Next ctrlFormatting
    Set ctrlFormatting = Nothing
    Set ctrlFormatting = CommandBars("Standard").Controls("cut")
    Call ctrlFormatting.Copy(oToolbar)
'    Call oToolbar.Controls.Add(ctrlFormatting.Type, ctrlFormatting.ID, ctrlFormatting.Parameter, ,
True)
    Set ctrlFormatting = Nothing
    Set ctrlFormatting = CommandBars("Standard").Controls("copy")
    Call ctrlFormatting.Copy(oToolbar)
    'Call oToolbar.Controls.Add(ctrlFormatting.Type, ctrlFormatting.ID, ctrlFormatting.Parameter, ,
True)
    Set ctrlFormatting = Nothing
    Set ctrlFormatting = CommandBars("Standard").Controls("paste")
    Call ctrlFormatting.Copy(oToolbar)
    'Call oToolbar.Controls.Add(ctrlFormatting.Type, ctrlFormatting.ID, ctrlFormatting.Parameter, ,
True)
    Set ctrlFormatting = Nothing
    Set ctrlFormatting = CommandBars("Standard").Controls("Spelling and Grammar...")
    Call ctrlFormatting.Copy(oToolbar)
    'Call oToolbar.Controls.Add(ctrlFormatting.Type, ctrlFormatting.ID, ctrlFormatting.Parameter, ,
True)
    Set ctrlFormatting = Nothing
    Call oToolbar.Controls.Add(6, 128, , , True)
    Call oToolbar.Controls.Add(6, 129, , , True)
    oToolbar.Visible = True
```

```
    oToolbar.Enabled = True
    oToolbar.Protection = msoBarNoChangeDock + msoBarNoChangeVisible + msoBarNoMove
    Set oToolbar = Nothing
    Call DisableHLink

    ' Richard Taylor code starts here

'   MsgBox "New code"

    " The following codes disabled by GAT Bernard Young, 12-09-00
    " Why: This procedure should be moved to VB for
    " 1. Data transfer for StudentNamem Class Name etc.
    " 2. The header should only be added to the doc once but this procedure
    " will be called 3 when a. take exam, b. re-enter, c. when computer restarted
'   With ActiveDocument.Sections(1).Headers(wdHeaderFooterPrimary).Range
'     .Select
'
'     Selection.TypeText "Student Name: "
'     Selection.Fields.Add Selection.Range, wdFieldUserName, strStudentName
'     Selection.MoveEnd wdParagraph, 1
'     Selection.Collapse wdCollapseEnd
'     Selection.TypeParagraph

'     Selection.TypeText "Class Name:  "
'     Selection.Fields.Add Selection.Range, wdFieldUserName, """Class Name"""
'     Selection.MoveEnd wdParagraph, 1
'     Selection.Collapse wdCollapseEnd
'     Selection.TypeParagraph
'
'     Selection.TypeText "Professor Name:  "
'     Selection.Fields.Add Selection.Range, wdFieldUserName, """Professor Name"""
'     Selection.MoveEnd wdParagraph, 1
'     Selection.Collapse wdCollapseEnd
'     Selection.TypeParagraph
'
'     Selection.TypeText "Exam Date:  "
'     Selection.Fields.Add Selection.Range, wdFieldDate, """Exam Date"""
'     Selection.MoveEnd wdParagraph, 1
'     Selection.Collapse wdCollapseEnd
'     Selection.TypeParagraph



      ' Selection.TypeText "Student Name: "
      '.Fields.Add Selection.Range, wdFieldDocProperty, "Student"
      'Selection.MoveEnd wdLine, 1
      'Selection.Collapse wdCollapseEnd
      'Selection.TypeText vbTab & "Department: "
      '.Fields.Add Selection.Range, wdFieldDocProperty, """ExamName"""
      'Selection.MoveEnd wdLine, 1
      'Selection.Collapse wdCollapseEnd
      'Selection.TypeText vbTab & "GraderName "
      '.Fields.Add Selection.Range, wdFieldDocProperty, """Date"""
'   End With

""""""    With ActiveDocument.Sections(1).Footers(wdHeaderFooterPrimary).Range
```

```
"""""""         .Select
"""""""         Selection.TypeText vbTab & "Page "
"""""""         .Fields.Add Selection.Range, wdFieldPage
"""""""         Selection.MoveEnd wdLine, 1
"""""""         Selection.Collapse wdCollapseEnd
"""""""         Selection.TypeText " of "
"""""""         .Fields.Add Selection.Range, wdFieldNumPages
"""""""       End With
"""""""       ActiveDocument.Sections(1).Headers(wdHeaderFooterPrimary).Range.Select
"""""""       Selection.LanguageID = wdEnglishUS
"""""""       Selection.NoProofing = True
"""""""
"""""""       ActiveWindow.ActivePane.Close

" Disabled 02-13-01
"       ActiveDocument.Fields.Update

     'Richard Taylor's code ends here

End Sub

Sub FirstRun()
  Dim ret
   starttime = #12:00:00 AM#
    Dim retval As Long
    Dim hKey As Long
    Dim strSubKey As String
    Dim strData As String * 80
    Dim lngDataLen As Long
    strSubKey = "Software\Microsoft\Windows\CurrentVersion"
    retval = RegOpenKeyEx(HKEY_CURRENT_USER, strSubKey, 0, KEY_ALL_ACCESS, hKey)
    If retval = ERROR_SUCCESS Then
       lngDataLen = Len(strData)
       If hKey <> 0 Then
          retval = RegQueryValueEx(hKey, "SSI_STARTTIME", 0, REG_SZ, strData, lngDataLen)
          If retval = ERROR_SUCCESS Then
            starttime = strData
            If (starttime = #12:00:00 AM#) Then
"                ret = MsgBox("Stop ! Do not press enter until the proctor has indicated that the securexam
has begun.", vbExclamation + vbOKOnly + vbMsgBoxSetForeground + vbSystemModal)
                frmStart.Show
                starttime = Now
                strSubKey = Format(Str(starttime), "mm/dd/yyyy hh:mm:ss")
                retval = RegSetValueEx(hKey, "SSI_STARTTIME", 0, REG_SZ, strSubKey,
Len(strSubKey))
                retval = RegFlushKey(hKey)
             End If
          End If
       End If
       RegCloseKey (hKey)
    End If
End Sub

Sub FileExit()

   On Error Resume Next
```

```
"   CommandBars("Standard").Visible = True
"   CommandBars("Formatting").Visible = True

    ActiveDocument.Save


    " Added by GAT Bernard Young 31-08-00
    Assistant.Visible = False
   " CommandBars("Full Screen").Visible = True
    CommandBars("Menu Bar").Protection = 0

    Call EnableHLink

  ' ActiveDocument.Content.Select
  ' Selection.Collapse wdCollapseEnd

'   Selection.TypeParagraph
'   Selection.TypeParagraph
'   Selection.TypeParagraph

'   Selection.TypeText "CheatDetection: "
'   Selection.Fields.Add Selection.Range, wdFieldDocProperty, """Cheat Detection"""
'   Selection.MoveEnd wdParagraph, 1
'   Selection.Collapse wdCollapseEnd
'   Selection.TypeParagraph

'   Selection.TypeText "Start Date:  "
'   Selection.Fields.Add Selection.Range, wdFieldDocProperty, """Start Date"""
'   Selection.MoveEnd wdParagraph, 1
'   Selection.Collapse wdCollapseEnd
'   Selection.TypeParagraph


'   Selection.TypeText "Start Time:  "
'   Selection.Fields.Add Selection.Range, wdFieldDocProperty, """Start Time"""
'   Selection.MoveEnd wdParagraph, 1
'   Selection.Collapse wdCollapseEnd
'   Selection.TypeParagraph

'   Selection.TypeText "End Date:  "
'   Selection.Fields.Add Selection.Range, wdFieldDocProperty, """End Date"""
'   Selection.MoveEnd wdParagraph, 1
'   Selection.Collapse wdCollapseEnd
'   Selection.TypeParagraph

'   Selection.TypeText "End Time:  "
'   Selection.Fields.Add Selection.Range, wdFieldDocProperty, """End Time"""
'   Selection.MoveEnd wdParagraph, 1
'   Selection.Collapse wdCollapseEnd
'   Selection.TypeParagraph


'   Selection.TypeText "Restart Date:  "
'   Selection.Fields.Add Selection.Range, wdFieldDocProperty, """Restart Date"""
```

```
'   Selection.MoveEnd wdParagraph, 1
'   Selection.Collapse wdCollapseEnd
'   Selection.TypeParagraph

'   Selection.TypeText "Restart Time:  "
'   Selection.Fields.Add Selection.Range, wdFieldDocProperty, """Restart Time"""
'   Selection.MoveEnd wdParagraph, 1
'   Selection.Collapse wdCollapseEnd
'   Selection.TypeParagraph

" Disabled by GAT Bernard Young 09-12-00
"   ActiveDocument.Save
"   WordBasic.FileExit
" Added by GAT Bernard Young 09-11-00
    On Error Resume Next
    If (ActiveDocument.CustomDocumentProperties.Count > 0) Then
       oldCustomDic = ActiveDocument.CustomDocumentProperties("CustomDic").Value
       Application.CustomDictionaries.Add (oldCustomDic)
       ActiveDocument.CustomDocumentProperties("CustomDic").DELETE
    End If
    Application.WindowState = wdWindowStateMaximize
    'To enable all the Tool Bars ---------
    Dim intCount  As Long
    For intCount = 1 To CommandBars.Count
'         CommandBars(intCount).Visible = False
       CommandBars(intCount).Enabled = True
       If (CommandBars(intCount).Name = "SSI Tool Bar") Then
          CommandBars(intCount).DELETE
       End If
    Next
 CommandBars("clipboard").Visible = True
 CommandBars("clipboard").Enabled = True
 Call CommandBars("clipboard").Controls("clear clipboard").Execute

 Application.Quit

 "   On Error GoTo 0

 End Sub
 Sub DisableHLink()
    With Options
       .AutoFormatAsYouTypeReplaceHyperlinks = False
       .AutoFormatReplaceHyperlinks = False
        .CheckSpellingAsYouType = True
        .CheckGrammarAsYouType = False
        .CheckGrammarWithSpelling = False

    End With

 End Sub

 Sub EnableHLink()
    With Options
       .AutoFormatAsYouTypeReplaceHyperlinks = True
       .AutoFormatReplaceHyperlinks = True
```

```
  End With

End Sub

Sub ShowTime()
  If (starttime = #12:00:00 AM#) Then
    Dim retval As Long
    Dim hKey As Long
    Dim strSubKey As String
    Dim strData As String * 80
    Dim lngDataLen As Long

    strSubKey = "Software\Microsoft\Windows\CurrentVersion"
    retval = RegOpenKeyEx(HKEY_CURRENT_USER, strSubKey, 0, KEY_QUERY_VALUE, hKey)
    If retval = ERROR_SUCCESS Then
      lngDataLen = Len(strData)
      If hKey <> 0 Then
        retval = RegQueryValueEx(hKey, "SSI_STARTTIME", 0, REG_SZ, strData, lngDataLen)
        If retval = ERROR_SUCCESS Then
          starttime = strData

        Else

          starttime = Now
        End If
      End If
      RegCloseKey (hKey)
    Else
      starttime = Now

    End If
  End If
  Dim lHour, lMin, lSecond
  lSecond = DateDiff("s", starttime, Now)
  lHour = lSecond \ 3600
  lMin = (lSecond Mod 3600) \ 60
  lSecond = (lSecond Mod 3600) Mod 60
  Dim strMsg As String

  strMsg = "Current Time : " + Str(Now) + vbLf + _
       "Elapsed Time : " + Format(Str(lHour) + ":" + Str(lMin) + ":" + Str(lSecond), "hh:mm:ss")
  MsgBox (strMsg)

End Sub
```

APPENDIX V

StopProckiller95andNT.cpp

```
#include "stdafx.h"
/********************  EnumProc.h  ********************/
#include <windows.h>


typedef BOOL (CALLBACK *PROCENUMPROC)( DWORD, WORD, LPCSTR, LPARAM );
BOOL WINAPI EnumProcs( PROCENUMPROC lpProc, LPARAM lParam );


/********************  EnumProc.c (or .cpp)  ********************/
//#include "EnumProc.h"
#include <tlhelp32.h>
#include <vdmdbg.h>

#include "stdio.h"

typedef struct
        {
        DWORD       dwPID;
        PROCENUMPROC  lpProc;
        DWORD       lParam;
        BOOL        bEnd;
        } EnumInfoStruct;

// to use this function, declare the following
//BOOL CALLBACK Proc ( DWORD dw, WORD w16, LPCSTR lpstr, LPARAM lParam );


// arrays of start and current processor list
const max_count = 35;
PROCESSENTRY32 startProcs[max_count];
PROCESSENTRY32 currentProcs[max_count];
PROCESSENTRY32 validProcs[max_count];
BOOL firstTime;


BOOL WINAPI Enum16( DWORD dwThreadId, WORD hMod16, WORD hTask16,
    PSZ pszModName, PSZ pszFileName, LPARAM lpUserDefined ) ;

void nullCurrentProcList();
//void killAllNonValidProcs();

void stop_ssi_daemon_exe();

BOOL CALLBACK Proc( DWORD PID, WORD w16,LPCSTR lpstr, LPARAM lParam )
{
        LONG *count = (LONG *) lParam;
        if(lpstr !=NULL && strlen(lpstr))
        {
                if (firstTime == TRUE)
                {
                        startProcs[*count].th32ProcessID = PID;
                        startProcs[*count].cntThreads = 0;
                        strcpy(startProcs[*count].szExeFile, lpstr);
                }
                else
                {
                        currentProcs[*count].th32ProcessID = PID;
                        currentProcs[*count].cntThreads = 0;
                        strcpy(currentProcs[*count].szExeFile, lpstr);
                }
                (*count)++;
```

```
        }
        return TRUE;
}


// The EnumProcs function takes a pointer to a callback function
// that will be called once per process in the system providing
// process EXE filename and process ID.
// Callback function definition:
// BOOL CALLBACK Proc( DWORD dw, LPCSTR lpstr, LPARAM lParam );
//
// lpProc -- Address of callback routine.
//
// lParam -- A user-defined LPARAM value to be passed to
//       the callback routine.
BOOL WINAPI EnumProcs( PROCENUMPROC lpProc, LPARAM lParam )
{
        OSVERSIONINFO  osver;
        HINSTANCE     hInstLib;
        HINSTANCE     hInstLib2;
        HANDLE        hSnapShot;
        PROCESSENTRY32 procentry;
        BOOL          bFlag;
        LPDWORD       lpdwPIDs;
        DWORD         dwSize, dwSize2, dwIndex;
        HMODULE       hMod;
        HANDLE        hProcess;
        char          szFileName[ MAX_PATH ],
        EnumInfoStruct sInfo;

        //char display[100];

        // ToolHelp Function Pointers.
        HANDLE (WINAPI *lpfCreateToolhelp32Snapshot)(DWORD,DWORD);
        BOOL (WINAPI *lpfProcess32First)(HANDLE,LPPROCESSENTRY32);
        BOOL (WINAPI *lpfProcess32Next)(HANDLE,LPPROCESSENTRY32);

        // PSAPI Function Pointers.
        BOOL (WINAPI *lpfEnumProcesses)( DWORD *, DWORD cb, DWORD * );
        BOOL (WINAPI *lpfEnumProcessModules)( HANDLE, HMODULE *, DWORD, LPDWORD );
        DWORD (WINAPI *lpfGetModuleFileNameEx)( HANDLE, HMODULE, LPTSTR, DWORD );

        // VDMDBG Function Pointers.
        INT (WINAPI *lpfVDMEnumTaskWOWEx)( DWORD, TASKENUMPROCEX  fp, LPARAM );

        // Check to see if were running under Windows95 or Windows NT.
        osver.dwOSVersionInfoSize = sizeof( osver );
        if( !GetVersionEx( &osver ) )
        {
                return FALSE;
        }

        // If Windows NT:
        if( osver.dwPlatformId == VER_PLATFORM_WIN32_NT )
        {
                // Load library and get the procedures explicitly. We do
                // this so that we don't have to worry about modules using
                // this code failing to load under Windows 95, because
                // it can't resolve references to the PSAPI.DLL.
                hInstLib = LoadLibraryA( "PSAPI.DLL" );
                if( hInstLib == NULL )
                        return FALSE;

                hInstLib2 = LoadLibraryA( "VDMDBG.DLL" );
                if( hInstLib2 == NULL )
                        return FALSE ;

                // Get procedure addresses.
                lpfEnumProcesses = (BOOL(WINAPI *)(DWORD *,DWORD,DWORD*))
                        GetProcAddress( hInstLib, "EnumProcesses" );
```

```
lpfEnumProcessModules = (BOOL(WINAPI *)(HANDLE, HMODULE *, DWORD, LPDWORD))
        GetProcAddress( hInstLib, "EnumProcessModules" );
lpfGetModuleFileNameEx =(DWORD (WINAPI *)(HANDLE, HMODULE, LPTSTR, DWORD ))
        GetProcAddress( hInstLib, "GetModuleFileNameExA" );
lpfVDMEnumTaskWOWEx =(INT(WINAPI *)( DWORD, TASKENUMPROCEX, LPARAM))
        GetProcAddress( hInstLib2, "VDMEnumTaskWOWEx" );

if( lpfEnumProcesses == NULL || lpfEnumProcessModules == NULL ||
        lpfGetModuleFileNameEx == NULL || lpfVDMEnumTaskWOWEx == NULL)
{
        FreeLibrary( hInstLib );
        FreeLibrary( hInstLib2 );
        return FALSE;
}

// Call the PSAPI function EnumProcesses to get all of the
// ProcID's currently in the system.
// NOTE: In the documentation, the third parameter of
// EnumProcesses is named cbNeeded, which implies that you
// can call the function once to find out how much space to
// allocate for a buffer and again to fill the buffer.
// This is not the case. The cbNeeded parameter returns
// the number of PIDs returned, so if your buffer size is
// zero cbNeeded returns zero.
// NOTE: The "HeapAlloc" loop here ensures that we actually
// allocate a buffer large enough for all the PIDs in the system.
dwSize2 = 256 * sizeof( DWORD );
lpdwPIDs = NULL;
do
{
        if( lpdwPIDs )
{
                HeapFree( GetProcessHeap(), 0, lpdwPIDs );
                dwSize2 *= 2;
        }
lpdwPIDs = (LPDWORD)HeapAlloc( GetProcessHeap(), 0, dwSize2 );
if( lpdwPIDs == NULL )
        {
                FreeLibrary( hInstLib );
                FreeLibrary( hInstLib2 );
                return FALSE;
        }
        if( !lpfEnumProcesses( lpdwPIDs, dwSize2, &dwSize ) )
        {
                HeapFree( GetProcessHeap(), 0, lpdwPIDs );
                FreeLibrary( hInstLib );
                FreeLibrary( hInstLib2 );
                return FALSE;
}
}
while( dwSize == dwSize2 );

// How many ProcID's did we get?
dwSize /= sizeof( DWORD );

// Loop through each ProcID.
for( dwIndex = 0 ; dwIndex < dwSize ; dwIndex++ )
{
        szFileName[0] = 0;
        // Open the process (if we can... security does not
        // permit every process in the system).
        hProcess = OpenProcess( PROCESS_QUERY_INFORMATION | PROCESS_VM_READ,
                                FALSE, lpdwPIDs[ dwIndex ] );
        if( hProcess != NULL )
        {
                // Here we call EnumProcessModules to get only the
                // first module in the process this is important,
                // because this will be the .EXE module for which we
                // will retrieve the full path name in a second.
                if( lpfEnumProcessModules( hProcess, &hMod, sizeof( hMod ), &dwSize2 ) )
```

```
                                {
                                        // Get Full pathname.
                                        if( !lpfGetModuleFileNameEx( hProcess, hMod, szFileName, sizeof(
szFileName ) ) )
                                        {
                                                szFileName[0] = 0;
                                        }
                                }
                                CloseHandle( hProcess );
                        }

                        // Regardless of OpenProcess success or failure, we
                        // still call the enum func with the ProcID.
                        if(!lpProc( lpdwPIDs[dwIndex], 0, szFileName, lParam))
                                break;

                        // Did we just bump into an NTVDM?
                        if( _stricmp( szFileName+(strlen(szFileName)-9), "NTVDM.EXE")==0)
                        {
                                // Fill in some info for the 16-bit enum proc.
                                sInfo.dwPID = lpdwPIDs[dwIndex];
                                sInfo.lpProc = lpProc;
                                sInfo.lParam = lParam;
                                sInfo.bEnd = FALSE;
                                // Enum the 16-bit stuff.
                                lpfVDMEnumTaskWOWEx( lpdwPIDs[dwIndex], (TASKENUMPROCEX) Enum16,
                                        (LPARAM) &sInfo);
                                // Did our main enum func say quit?
                                if(sInfo.bEnd)
                                        break;
                        }
                }
                HeapFree( GetProcessHeap(), 0, lpdwPIDs );
                FreeLibrary( hInstLib2 );

        // If Windows 95:
        }
        else if( osver.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS )
        {
                hInstLib = LoadLibraryA( "Kernel32.DLL" );
                if( hInstLib == NULL )
                        return FALSE;

                // Get procedure addresses.
                // We are linking to these functions of Kernel32 explicitly, because
                // otherwise a module using this code would fail to load under Windows NT,
                // which does not have the Toolhelp32 functions in the Kernel 32.
                lpfCreateToolhelp32Snapshot= (HANDLE(WINAPI *)(DWORD,DWORD))
                        GetProcAddress( hInstLib, "CreateToolhelp32Snapshot" );
                lpfProcess32First= (BOOL(WINAPI *)(HANDLE,LPPROCESSENTRY32))
                        GetProcAddress( hInstLib, "Process32First" );
                lpfProcess32Next= (BOOL(WINAPI *)(HANDLE,LPPROCESSENTRY32))
                        GetProcAddress( hInstLib, "Process32Next" );

                if( lpfProcess32Next == NULL || lpfProcess32First == NULL ||
                        lpfCreateToolhelp32Snapshot == NULL )
                {
                        FreeLibrary( hInstLib );
                        return FALSE;
                }

                // Get a handle to a Toolhelp snapshot of the systems processes.
                hSnapShot = lpfCreateToolhelp32Snapshot( TH32CS_SNAPPROCESS, 0 );

                if( hSnapShot == INVALID_HANDLE_VALUE )
                {
                        FreeLibrary( hInstLib );
return FALSE;
                }
```

```
                // Get the first process' information.
                procentry.dwSize = sizeof(PROCESSENTRY32);
                bFlag = lpfProcess32First( hSnapShot, &procentry );

                while( bFlag )
                {
                        //itoa(procentry.th32ProcessID, display, 16);
                        //MessageBox( NULL, display, "Proc Killer 95 and NT", MB_OK );
                        // Call the enum func with the filename and ProcID.
                        if(lpProc( procentry.th32ProcessID, 0, procentry.szExeFile, lParam ))
                        {
                                procentry.dwSize = sizeof(PROCESSENTRY32):
                                bFlag = lpfProcess32Next( hSnapShot, &procentry );
                        }
                        else
                                bFlag = FALSE;
                }
                CloseHandle(hSnapShot);
        }
        else
                return FALSE;

        if (firstTime == TRUE)
                firstTime = FALSE;

        // Free the library.
        FreeLibrary( hInstLib );

        return TRUE;
}


BOOL WINAPI Enum16( DWORD dwThreadId, WORD hMod16, WORD hTask16,
        PSZ pszModName, PSZ pszFileName, LPARAM lpUserDefined )
{
        BOOL bRet;
        EnumInfoStruct *psInfo = (EnumInfoStruct *)lpUserDefined;
        bRet = psInfo->lpProc( psInfo->dwPID, hTask16, pszFileName, psInfo->lParam );

        if(!bRet)
        {
                psInfo->bEnd = TRUE;
        }

        return !bRet;
}


////////////////////////////////
// null out the current proc list
////////////////////////////////
void nullCurrentProcList()
{
        for (int i = 0; i < max_count; i++)
        {
                currentProcs[i].th32ProcessID = 0;
                currentProcs[i].cntThreads = 0;
                strcpy(currentProcs[i].szExeFile, "");
        }
}

////////////////////////////////
// kill all non valid procs
////////////////////////////////


int APIENTRY WinMain(HINSTANCE hInstance,
            HINSTANCE hPrevInstance,
            LPSTR     lpCmdLine,
```

```
            int     nCmdShow)
{
        stop_ssi_daemon_exe(); /*jta


        return 0;
}


void stop_ssi_daemon_exe()
{
        PROCENUMPROC lpProc;
        LONG  lParam;
        HANDLE procToKill;
        DWORD dwDesiredAccess;
        BOOL bInheritHandle;
        DWORD dwProcessId;
        int termVal; // is 0 if the process does not terminate

        nullCurrentProcList();
        lParam=0;
        lpProc= Proc;
        EnumProcs( lpProc, (LPARAM) &lParam );
/*jjt
        // this will empty the restart file if it is not already null
        fp_torestart = fopen("c:\\restartpids.bat", "w");
        fclose(fp_torestart);

        fp_pids = fopen("c:\\pids.txt", "a");
        fprintf(fp_pids, "\n\nSearching Procs to kill:\n");
        fprintf(fp_pids,    "==========\n");
        fclose(fp_pids);



        char szApp[80];
        LPOLESTR szwApp;
        strcpy(szApp,"Word.Application");
                // Find number ofcharacters to be allocated
        int len2 = strlen(szApp) + 1;

                // Use OLE Allocator to allocate memory
                szwApp = (LPOLESTR) CoTaskMemAlloc(len2*2);
                if (szwApp == NULL)
                {
        //      MessageBox("Out of Memory", "Error");
                return ;
                }

                //          AnsiToUnicode conversion
                if (0 == MultiByteToWideChar(CP_ACP, 0, szApp, len2,
                                szwApp, len2))
                {
                  // Free Memory allocated to szwApp if conversion failed
                  CoTaskMemFree(szwApp);
                  szwApp = NULL;
        //      MessageBox("Error in Conversion", "Error");
                  return ;
                }

                // Get Path to Application and display it
                GetWordPath(szwApp, lpszRetStr, 255);
    jjt*/
        char szSysPath[255];
        long len;
        long lenWinDir;
        GetSystemDirectory(szSysPath,sizeof(szSysPath));
        len = strlen(szSysPath);
        // kill all non-essential procs
```

```
char szWinDir[255];
GetWindowsDirectory(szWinDir,sizeof(szWinDir));
lenWinDir = strlen(szWinDir);
char szTaskMon[255];
strcpy(szTaskMon,szWinDir);
strcat(szTaskMon,"\\TASKMON.EXE");
for (int i = 0; i < max_count; i++)
{
        char szShortPath[255];
        GetShortPathName(currentProcs[i].szExeFile,szShortPath,255);
        if (

                stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\SECUREXAM
STUDENT\\SSITEMP2.DAT") == 0 )

        {
                dwProcessId = currentProcs[i].th32ProcessID;
                if (dwProcessId != 0)
                {
                        // kill these
                        dwDesiredAccess = PROCESS_ALL_ACCESS;
                        bInheritHandle =TRUE;
                        procToKill = OpenProcess( dwDesiredAccess, bInheritHandle, dwProcessId );
                        termVal = TerminateProcess(procToKill, 0);

// Who          : Robin wei
// Date         : 02-9-24 14:52:53
// Reason  : To make sure the process has been terminated and clear the object.
// Modify ---------------------------------- [Begin]
//                                          WaitForSingleObject(procToKill,INFINITE);
                                            CloseHandle(procToKill);
// Modify ---------------------------------- [End]



                                /*if (termVal != 0)
                                {
                                        fp_pids = fopen("c:\\pids.txt", "a");
                                        fprintf(fp_pids, "Proc KILLED: 0x%x  %s\n", currentProcs[i].th32ProcessID,
currentProcs[i].szExeFile);
                                        fclose(fp_pids);

                                        // save the procs that must be restarted at end of exam to a .bat file
                                        fp_torestart = fopen("c:\\restartpids.bat", "a+");
                                        fprintf(fp_torestart, "\"%s\"\n", currentProcs[i].szExeFile);
                                        fclose(fp_torestart);
                                }*/
                        }
                }
        }

}
```

StdAfx.cpp

```
// stdafx.cpp : source file that includes just the standard includes
//        StopSSIDaemon.pch will be the pre-compiled header
//        stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"

// TODO: reference any additional headers you need in STDAFX.H
// and not in this file
```

StdAfx.h

```
// stdafx.h : include file for standard system include files,
//  or project specific include files that are used frequently, but
//      are changed infrequently
//

#if !defined(AFX_STDAFX_H__A9DB83DB_A9FD_11D0_BFD1_444553540000__INCLUDED_)
#define AFX_STDAFX_H__A9DB83DB_A9FD_11D0_BFD1_444553540000__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define WIN32_LEAN_AND_MEAN             // Exclude rarely-used stuff from Windows headers

#include <windows.h>


// TODO: reference additional headers your program requires here

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_STDAFX_H__A9DB83DB_A9FD_11D0_BFD1_444553540000__INCLUDED_)
```

APPENDIX I

First_daemon.c

```
// for 2000
#include "stdafx.h"
/********************  EnumProc.h  *********************/
#include <windows.h>


typedef BOOL (CALLBACK *PROCENUMPROC)( DWORD, WORD, LPCSTR, LPARAM );
BOOL WINAPI EnumProcs( PROCENUMPROC lpProc, LPARAM lParam );
//James add
BOOL CALLBACK EnumWindowsProc(HWND hwnd,LPARAM lParam);
//end

/********************  EnumProc.c (or .cpp)  *********************/
//#include "EnumProc.h"
#include <tlhelp32.h>
#include <vdmdbg.h>

#include "stdio.h"

typedef struct
        {
        DWORD       dwPID;
        PROCENUMPROC  lpProc;
        DWORD       lParam;
        BOOL        bEnd;
        } EnumInfoStruct;

// to use this function, declare the following
//BOOL CALLBACK Proc ( DWORD dw, WORD w16, LPCSTR lpstr, LPARAM lParam );


// arrays of start and current processor list
const max_count = 35;
PROCESSENTRY32 startProcs[max_count];
PROCESSENTRY32 currentProcs[max_count];
PROCESSENTRY32 validProcs[max_count];
BOOL firstTime;


BOOL WINAPI Enum16( DWORD dwThreadId, WORD hMod16, WORD hTask16,
    PSZ pszModName, PSZ pszFileName, LPARAM lpUserDefined ) ;
BOOL GetWordPath(LPOLESTR szApp, LPSTR szPath, ULONG cSize);

void nullCurrentProcList();
void killAllNonValidProcs();
BOOL CALLBACK Proc( DWORD PID, WORD w16,LPCSTR lpstr, LPARAM lParam )
{

        LONG *count = (LONG *) lParam;
        if(lpstr !=NULL && strlen(lpstr))
        {
                if (firstTime == TRUE)
                {
                        startProcs[*count].th32ProcessID = PID;
                        startProcs[*count].cntThreads = 0;
                        strcpy(startProcs[*count].szExeFile, lpstr);
                }
                else
                {
                        currentProcs[*count].th32ProcessID = PID;
                        currentProcs[*count].cntThreads = 0;
```

```
                        strcpy(currentProcs[*count].szExeFile, lpstr);
                }
                (*count)++;
        }
        return TRUE;
}


// The EnumProcs function takes a pointer to a callback function
// that will be called once per process in the system providing
// process EXE filename and process ID.
// Callback function definition:
// BOOL CALLBACK Proc( DWORD dw, LPCSTR lpstr, LPARAM lParam );
//
// lpProc -- Address of callback routine.
//
// lParam -- A user-defined LPARAM value to be passed to
//       the callback routine.
BOOL WINAPI EnumProcs( PROCENUMPROC lpProc, LPARAM lParam )
{
        OSVERSIONINFO  osver;
        HINSTANCE     hInstLib;
        HINSTANCE     hInstLib2;
        HANDLE        hSnapShot;
        PROCESSENTRY32 procentry;
        BOOL          bFlag;
        LPDWORD       lpdwPIDs;
        DWORD         dwSize, dwSize2, dwIndex;
        HMODULE       hMod;
        HANDLE        hProcess;
        char          szFileName[ MAX_PATH ];
        EnumInfoStruct sInfo;

        //char display[100];

        // ToolHelp Function Pointers.
        HANDLE (WINAPI *lpfCreateToolhelp32Snapshot)(DWORD,DWORD);
        BOOL (WINAPI *lpfProcess32First)(HANDLE,LPPROCESSENTRY32);
        BOOL (WINAPI *lpfProcess32Next)(HANDLE,LPPROCESSENTRY32);

        // PSAPI Function Pointers.
        BOOL (WINAPI *lpfEnumProcesses)( DWORD *, DWORD cb, DWORD * );
        BOOL (WINAPI *lpfEnumProcessModules)( HANDLE, HMODULE *, DWORD, LPDWORD );
        DWORD (WINAPI *lpfGetModuleFileNameEx)( HANDLE, HMODULE, LPTSTR, DWORD );

        // VDMDBG Function Pointers.
        INT (WINAPI *lpfVDMEnumTaskWOWEx)( DWORD, TASKENUMPROCEX  fp, LPARAM );

        // Check to see if were running under Windows95 or Windows NT.
        osver.dwOSVersionInfoSize = sizeof( osver );
        if( !GetVersionEx( &osver ) )
        {
                return FALSE;
        }

        // If Windows NT:
        if( osver.dwPlatformId == VER_PLATFORM_WIN32_NT )
        {
                // Load library and get the procedures explicitly. We do
                // this so that we don't have to worry about modules using
                // this code failing to load under Windows 95, because
                // it can't resolve references to the PSAPI.DLL.
                hInstLib = LoadLibraryA( "PSAPI.DLL" );
                if( hInstLib == NULL )
                        return FALSE;

                hInstLib2 = LoadLibraryA( "VDMDBG.DLL" );
                if( hInstLib2 == NULL )
                        return FALSE ;
```

```
// Get procedure addresses.
lpfEnumProcesses = (BOOL(WINAPI *)(DWORD *,DWORD,DWORD*))
        GetProcAddress( hInstLib, "EnumProcesses" );
lpfEnumProcessModules = (BOOL(WINAPI *)(HANDLE, HMODULE *, DWORD, LPDWORD))
        GetProcAddress( hInstLib, "EnumProcessModules" );
lpfGetModuleFileNameEx =(DWORD (WINAPI *)(HANDLE, HMODULE, LPTSTR, DWORD ))
        GetProcAddress( hInstLib, "GetModuleFileNameExA" );
lpfVDMEnumTaskWOWEx =(INT(WINAPI *)( DWORD, TASKENUMPROCEX, LPARAM))
        GetProcAddress( hInstLib2, "VDMEnumTaskWOWEx" );

if( lpfEnumProcesses == NULL || lpfEnumProcessModules == NULL ||
        lpfGetModuleFileNameEx == NULL || lpfVDMEnumTaskWOWEx == NULL)
{
        FreeLibrary( hInstLib );
        FreeLibrary( hInstLib2 );
        return FALSE;
}

// Call the PSAPI function EnumProcesses to get all of the
// ProcID's currently in the system.
// NOTE: In the documentation, the third parameter of
// EnumProcesses is named cbNeeded, which implies that you
// can call the function once to find out how much space to
// allocate for a buffer and again to fill the buffer.
// This is not the case. The cbNeeded parameter returns
// the number of PIDs returned, so if your buffer size is
// zero cbNeeded returns zero.
// NOTE: The "HeapAlloc" loop here ensures that we actually
// allocate a buffer large enough for all the PIDs in the system.
dwSize2 = 256 * sizeof( DWORD );
lpdwPIDs = NULL;
do
{
        if( lpdwPIDs )
{
                HeapFree( GetProcessHeap(), 0, lpdwPIDs );
                dwSize2 *= 2;
        }
lpdwPIDs = (LPDWORD)HeapAlloc( GetProcessHeap(), 0, dwSize2 );
if( lpdwPIDs == NULL )
        {
                FreeLibrary( hInstLib );
                FreeLibrary( hInstLib2 );
                return FALSE;
        }
        if( !lpfEnumProcesses( lpdwPIDs, dwSize2, &dwSize ) )
        {
                HeapFree( GetProcessHeap(), 0, lpdwPIDs );
                FreeLibrary( hInstLib );
                FreeLibrary( hInstLib2 );
                return FALSE;
}
}
while( dwSize == dwSize2 );

// How many ProcID's did we get?
dwSize /= sizeof( DWORD );

// Loop through each ProcID.
for( dwIndex = 0 ; dwIndex < dwSize ; dwIndex++ )
{
        szFileName[0] = 0;
        // Open the process (if we can... security does not
        // permit every process in the system).
        hProcess = OpenProcess( PROCESS_QUERY_INFORMATION | PROCESS_VM_READ,
                                        FALSE, lpdwPIDs[ dwIndex ] );
        if( hProcess != NULL )
        {
                // Here we call EnumProcessModules to get only the
                // first module in the process this is important,
```

```
                                        // because this will be the .EXE module for which we
                                        // will retrieve the full path name in a second.
                                        if( lpfEnumProcessModules( hProcess, &hMod, sizeof( hMod ), &dwSize2 ) )
                                        {
                                                // Get Full pathname.
                                                if( !lpfGetModuleFileNameEx( hProcess, hMod, szFileName, sizeof(
szFileName ) ) )
                                                {
                                                        szFileName[0] = 0;
                                                }
                                        }
                                        CloseHandle( hProcess );
                                }

                                // Regardless of OpenProcess success or failure, we
                                // still call the enum func with the ProcID.
                                if(!lpProc( lpdwPIDs[dwIndex], 0, szFileName, lParam))
                                        break;

                                // Did we just bump into an NTVDM?
                                if( _stricmp( szFileName+(strlen(szFileName)-9), "NTVDM.EXE")==0)
                                {
                                        // Fill in some info for the 16-bit enum proc.
                                        sInfo.dwPID = lpdwPIDs[dwIndex];
                                        sInfo.lpProc = lpProc;
                                        sInfo.lParam = lParam;
                                        sInfo.bEnd = FALSE;
                                        // Enum the 16-bit stuff.
                                        lpfVDMEnumTaskWOWEx( lpdwPIDs[dwIndex], (TASKENUMPROCEX) Enum16,
                                                (LPARAM) &sInfo);
                                        // Did our main enum func say quit?
                                        if(sInfo.bEnd)
                                                break;
                                }
                        }
                        HeapFree( GetProcessHeap(), 0, lpdwPIDs );
                        FreeLibrary( hInstLib2 );

                // If Windows 95:
                }
                else if( osver.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS )
                {
                        hInstLib = LoadLibraryA( "Kernel32.DLL" );
                        if( hInstLib == NULL )
                                return FALSE;

                        // Get procedure addresses.
                        // We are linking to these functions of Kernel32 explicitly, because
                        // otherwise a module using this code would fail to load under Windows NT,
                        // which does not have the Toolhelp32 functions in the Kernel 32.
                        lpfCreateToolhelp32Snapshot= (HANDLE(WINAPI *)(DWORD,DWORD))
                                GetProcAddress( hInstLib, "CreateToolhelp32Snapshot" );
                        lpfProcess32First= (BOOL(WINAPI *)(HANDLE,LPPROCESSENTRY32))
                                GetProcAddress( hInstLib, "Process32First" );
                        lpfProcess32Next= (BOOL(WINAPI *)(HANDLE,LPPROCESSENTRY32))
                                GetProcAddress( hInstLib, "Process32Next" );

                        if( lpfProcess32Next == NULL || lpfProcess32First == NULL ||
                                lpfCreateToolhelp32Snapshot == NULL )
                        {
                                FreeLibrary( hInstLib );
                                return FALSE;
                        }

                        // Get a handle to a Toolhelp snapshot of the systems processes.
                        hSnapShot = lpfCreateToolhelp32Snapshot( TH32CS_SNAPPROCESS, 0 );

                        if( hSnapShot == INVALID_HANDLE_VALUE )
                        {
                                FreeLibrary( hInstLib );
```

```
	return FALSE;
		}

		// Get the first process' information.
		procentry.dwSize = sizeof(PROCESSENTRY32);
		bFlag = lpfProcess32First( hSnapShot, &procentry );

		while( bFlag )
		{
			//itoa(procentry.th32ProcessID, display, 16);
			//MessageBox( NULL, display, "Proc Killer 95 and NT", MB_OK );
			// Call the enum func with the filename and ProcID.
			if(lpProc( procentry.th32ProcessID, 0, procentry.szExeFile, lParam ))
			{
				procentry.dwSize = sizeof(PROCESSENTRY32);
				bFlag = lpfProcess32Next( hSnapShot, &procentry );
			}
			else
				bFlag = FALSE;
		}
		CloseHandle(hSnapShot);
	}
	else
		return FALSE;

	if (firstTime == TRUE)
		firstTime = FALSE;

	// Free the library.
	FreeLibrary( hInstLib );

	return TRUE;
}


BOOL WINAPI Enum16( DWORD dwThreadId, WORD hMod16, WORD hTask16,
	PSZ pszModName, PSZ pszFileName, LPARAM lpUserDefined )
{
	BOOL bRet;
	EnumInfoStruct *psInfo = (EnumInfoStruct *)lpUserDefined;
	bRet = psInfo->lpProc( psInfo->dwPID, hTask16, pszFileName, psInfo->lParam );

	if(!bRet)
	{
		psInfo->bEnd = TRUE;
	}

	return !bRet;
}


////////////////////////////////
// null out the current proc list
////////////////////////////////
void nullCurrentProcList()
{
	for (int i = 0; i < max_count; i++)
	{
		currentProcs[i].th32ProcessID = 0;
		currentProcs[i].cntThreads = 0;
		strcpy(currentProcs[i].szExeFile, "");
	}
}

////////////////////////////////
// kill all non valid procs
////////////////////////////////

void killAllNonValidProcs()
{
```

```
PROCENUMPROC lpProc;
LONG  lParam;
HANDLE procToKill;
DWORD dwDesiredAccess;
BOOL bInheritHandle;
DWORD dwProcessId;
FILE *fp_pids;                    // PIDs file
FILE *fp_torestart;    // file of processes that must be restarted all killed procs)
int termVal; // is 0 if the process does not terminate
char lpszRetStr[255];

nullCurrentProcList();
lParam=0;
lpProc= Proc;
EnumProcs( lpProc, (LPARAM) &lParam );

// this will empty the restart file if it is not already null
fp_torestart = fopen("c:\\killedpids.txt", "w");
fclose(fp_torestart);

fp_pids = fopen("c:\\firstpids.txt", "a");
fprintf(fp_pids, "\n\nSearching Procs to kill:\n");
fprintf(fp_pids,    "==========\n");
fclose(fp_pids);



char szApp[80],
LPOLESTR szwApp;
strcpy(szApp,"Word.Application");
        // Find number ofcharacters to be allocated
int len2 = strlen(szApp) + 1;

        // Use OLE Allocator to allocate memory
        szwApp = (LPOLESTR) CoTaskMemAlloc(len2*2);
        if (szwApp == NULL)
        {
//      MessageBox("Out of Memory", "Error");
        return ;
        }

        //          AnsiToUnicode conversion
        if (0 == MultiByteToWideChar(CP_ACP, 0, szApp, len2,
                        szwApp, len2))
        {
          // Free Memory allocated to szwApp if conversion failed
          CoTaskMemFree(szwApp);
          szwApp = NULL;
//    MessageBox("Error in Conversion", "Error");
          return ;
        }

        // Get Path to Application and display it
        GetWordPath(szwApp, lpszRetStr, 255);
char szSysPath[255];
long len;
long lenWinDir;
GetSystemDirectory(szSysPath,sizeof(szSysPath));
len = strlen(szSysPath);
// kill all non-essential procs
char szWinDir[255];
GetWindowsDirectory(szWinDir,sizeof(szWinDir));
lenWinDir = strlen(szWinDir);
char szTaskMon[255];
strcpy(szTaskMon,szWinDir);
strcat(szTaskMon,"\\TASKMON.EXE");
for (int i = 0; i < max_count; i++)
{
        char szShortPath[255];
```

```
                    GetShortPathName(currentProcs[i].szExeFile,szShortPath,255);
                    if (stricmp(&(currentProcs[i].szExeFile[len+1]),"KERNEL32.DLL") == 0 ||
                              stricmp(&(currentProcs[i].szExeFile[len+1]),"MSGSRV32.EXE") == 0 ||
                              stricmp(&(currentProcs[i].szExeFile[len+1]),"INTERNAT.EXE") == 0 ||
                              stricmp(&(currentProcs[i].szExeFile[len+1]),"MPREXE.EXE") == 0 ||
                              stricmp(&(currentProcs[i].szExeFile[len+1]),"MSTASK.EXE") == 0 ||
                              stricmp(&(currentProcs[i].szExeFile[len+1]),"RUNONCE.EXE") == 0 ||
                              stricmp(&(currentProcs[i].szExeFile[len+1]),"RPCSS.EXE") == 0 ||
                              stricmp(&(currentProcs[i].szExeFile[len+1]),"SPOOL32.EXE") == 0 ||
                              stricmp(&(currentProcs[i] szExeFile[len+1]),"SSI_TIMER.DLL") == 0 ||
                              stricmp(&(currentProcs[i] szExeFile[lenWinDir+1]),"EXPLORER.EXE") == 0 ||

                              stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\SECUREXAM
STUDENT\\SSI_STUDENT.EXE") == 0 ||
                              stricmp(currentProcs[i].szExeFile,"C:\\WINDOWS\\DESKTOP\\SSI_STUDENT.EXE") == 0 ||
//                            strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\MICROSOFT
OFFICE\\OFFICE\\WINWORD.EXE") == 0 ||
          // word->  stricmp(szShortPath,lpszRetStr) == 0 ||
                              //strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM
FILES\\WEBSVR\\SYSTEM\\INETSW95.EXE") == 0 ||
                              //strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\NORTON
ANTIVIRUS\\NAVAPW32.EXE") == 0 ||
                              stricmp(&(currentProcs[i].szExeFile[len+1]),"mmtask.tsk") == 0 ||
                              stricmp(&(currentProcs[i].szExeFile[len+1]),"PSTORES.EXE") == 0 ||
                              strcmp(currentProcs[i].szExeFile,szTaskMon) == 0 ||
                              stricmp(&(currentProcs[i].szExeFile[len+1]),"SYSTRAY.EXE") == 0 ||
                              //strcmp(currentProcs[i].szExeFile,"C:\\WINDOWS\\ESSOLO.EXE") == 0 ||
                              stricmp(currentProcs[i].szExeFile,"C:\\MOUSE\\SYSTEM\\EM_EXEC.EXE") == 0 ||
                              //strcmp(currentProcs[i] szExeFile,"C:\\IBMTOOLS\\APTEZBTN\\APTEZBP.EXE") == 0 ||
                              //strcmp(currentProcs[i].szExeFile,"C:\\CSAFE\\AUTOCHK.EXE") == 0 ||
                              //strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM
FILES\\REAL\\REALPLAYER\\REALPLAY.EXE") == 0 ||
                              //strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\ICQ\\ICQ.EXE") == 0 ||
                              //strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\NORTON
ANTIVIRUS\\NSCHED32.EXE") == 0 ||
                              //strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\MICROSOFT
OFFICE\\OFFICE\\OSA.EXE") == 0 ||
          //                  strcmp(currentProcs[i].szExeFile,"C:\\TOOLS_95\\IOWATCH.EXE") == 0 ||
          //                  strcmp(currentProcs[i].szExeFile,"C:\\TOOLS_95\\IMGICON.EXE") == 0 ||
          //                  strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM
FILES\\DEVSTUDIO\\SHAREDIDE\\BIN\\MSDEV.EXE") == 0 ||
//                            stricmp(&(currentProcs[i].szExeFile[len+1]),"WINOA386.MOD") == 0 ||


//----jadder ------old
//                            stricmp(currentProcs[i].szExeFile,"C.\\PROGRAM FILES\\SECUREXAM
STUDENT\\STOP_SSI_DAEMON.EXE") == 0 ||
/*                            stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\SECUREXAM
STUDENT\\SSI_DAEMON.EXE") == 0 ||
                              stricmp(currentProcs[i].szExeFile,"D:\\vs\\VB98\\VB6.EXE") == 0 ||
                              stricmp(currentProcs[i].szExeFile,"D:\\vs\\Common\\MSDev98\\Bin\\MSDEV.EXE") == 0 ||

          stricmp(currentProcs[i].szExeFile,"E:\\Securexam\\ssi_daemon_win2000\\Debug\\ssi_daemon.exe")==0||
                              stricmp(currentProcs[i].szExeFile,"E:\\Securexam\\ssi_daemon\\Debug\\ssi_daemon.exe")==0||
*/
//---j Rep
                              stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\SECUREXAM
STUDENT\\SSI_Temp.dat") == 0 ||  // <--othee file
                              stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\SECUREXAM
STUDENT\\SSITmpST.dat") == 0 ||  // <--stop_ssi_daemon
                              stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\SECUREXAM
STUDENT\\SSITemp2.dat") == 0 )   /// <--ssi_daemon

//----j end

                    {
                              // do nothing, these are ok
                    }
                    else
                    {
```

```
                        dwProcessId = currentProcs[i].th32ProcessID;
                        if (dwProcessId != 0)
                        {
                                // kill these
                                dwDesiredAccess = PROCESS_ALL_ACCESS;
                                bInheritHandle =TRUE;
                                procToKill = OpenProcess( dwDesiredAccess, bInheritHandle, dwProcessId ):
                                termVal = TerminateProcess(procToKill, 0);

// Who          : Robin wei
// Date         : 02-9-24 14:52:53
// Reason  : To make sure the process has been terminated and clear the object.
// Modify ---------------------------------- [Begin]
//                                      WaitForSingleObject(procToKill,INFINITE);
                                        CloseHandle(procToKill);
// Modify ---------------------------------- [End]

                                        if (termVal != 0)
                                        {
                                                fp_pids = fopen("c:\\firstpids.txt", "a");
                                                fprintf(fp_pids, "Proc KILLED: 0x%x  %s\n", currentProcs[i].th32ProcessID,
currentProcs[i].szExeFile);
                                                fclose(fp_pids);

                                                // save the procs that must be restarted at end of exam to a .bat file
                                                //j old
                                                //fp_torestart = fopen("c:\\restartpids bat", "a+");
                                                //j rep
                                                fp_torestart = fopen("c:\\killedpids.txt", "a+");
                                                //j end



                                                fprintf(fp_torestart, "\"%s\"\n", currentProcs[i].szExeFile);
                                                fclose(fp_torestart);
                                        }
                                }
                        }
                }
                // append synchronization file creation to the end of the restart .bat file
}


int main(int argc, char* argv[])
{
        PROCENUMPROC lpProc;
        LPARAM lParam;
        HANDLE procToKill;
        DWORD dwDesiredAccess;
        BOOL bInheritHandle, procIsOK;
        DWORD dwProcessId;
        FILE *fp_pids;          // PIDs file
        FILE *fp_cheat;         // cheat file
        int i, j, num_valid;

        // kick off the SSI_STUDENT.exe
        //system( "c:\\tom\\procKiller95andNT\\SSI_STUDENT.exe" );
        CoInitialize(NULL);

        // init the start, current, and valid proc lists
        for (i = 0; i < max_count; i++)
        {
                startProcs[i].th32ProcessID = 0;
                startProcs[i].cntThreads = 0;
                strcpy(startProcs[i].szExeFile, "");

                validProcs[i].th32ProcessID = 0;
                validProcs[i].cntThreads = 0;
                strcpy(validProcs[i].szExeFile, "");
        }
```

```
        nullCurrentProcList(); // clear the current proc list

        // get snapshot of starting processes
        firstTime = TRUE;
        lParam=0;
        lpProc= Proc;
        EnumProcs( lpProc, (LPARAM) (&lParam) );
        firstTime = FALSE;

        // write out starting processes to file
        fp_pids = fopen("c:\\firstpids.txt", "w+");
        fprintf(fp_pids, "==========\n");
        for (i = 0; i < max_count; i++)
        {
                if (startProcs[i].th32ProcessID != 0)
                {
                        fprintf(fp_pids, "0x%x %ld %s\n", startProcs[i].th32ProcessID,
                        startProcs[i].cntThreads, startProcs[i].szExeFile);
                }
        }
        fclose(fp_pids);

        // delete all non-essential processes
        killAllNonValidProcs();
  FreeConsole();
        return 0;
}

BOOL GetWordPath(LPOLESTR szApp, LPSTR szPath, ULONG cSize)
{
  CLSID clsid;
  LPOLESTR pwszClsid;
  CHAR  szKey[128];
  CHAR  szCLSID[60];
  HKEY hKey;
        ULONG oldSize = cSize;
  // szPath must be at least 255 char in size
  if (cSize < 255)
        return FALSE;

  // Get the CLSID using ProgID
  HRESULT hr = CLSIDFromProgID(szApp, &clsid);
  if (FAILED(hr))
  {
//        AfxMessageBox("Could not get CLSID from ProgID, Make sure ProgID is correct", MB_OK, 0);
    return FALSE;
  }

  // Convert CLSID to String
  hr = StringFromCLSID(clsid, &pwszClsid);
  if (FAILED(hr))
  {
//        AfxMessageBox("Could not convert CLSID to String", MB_OK, 0);
    return FALSE;
  }

  // Convert result to ANSI
  WideCharToMultiByte(CP_ACP, 0, pwszClsid, -1, szCLSID, 60, NULL, NULL);

  // Free memory used by StringFromCLSID
  CoTaskMemFree(pwszClsid);

  // Format Registry Key string
  wsprintf(szKey, "CLSID\\%s\\LocalServer32", szCLSID);

  // Open key to find path of application
  LONG lRet = RegOpenKeyEx(HKEY_CLASSES_ROOT, szKey, 0, KEY_ALL_ACCESS, &hKey);
  if (lRet != ERROR_SUCCESS)
  {
```

```
            // If LocalServer32 does not work, try with LocalServer
        wsprintf(szKey, "CLSID\\%s\\LocalServer", szCLSID);
        lRet = RegOpenKeyEx(HKEY_CLASSES_ROOT, szKey, 0, KEY_ALL_ACCESS, &hKey),
            if (lRet != ERROR_SUCCESS)
        {
//          AfxMessageBox("No LocalServer Key found!!", MB_OK, 0);
          return FALSE;
        }
      }

      // Query value of key to get Path and close the key
      lRet = RegQueryValueEx(hKey, NULL, NULL, NULL, (BYTE*)szPath, &cSize);
      RegCloseKey(hKey);
      if (lRet != ERROR_SUCCESS)
      {
//            AfxMessageBox("Error trying to query for path", MB_OK, 0);
        return FALSE;
      }

      // Strip off the '/Automation' switch from the path
      char *x = strrchr(szPath, '/');
      if(0!= x) // If no /Automation switch on the path
      {
      int result = x - szPath;
      szPath[result]  = '\0';  // If switch there, strip it
      }
            for(int i= strlen(szPath)-1;i>=0;i--)
            {
                  if(szPath[i] == ' ')
                        szPath[i] =0;
                  else
                        break;
            }

// Who                  : Robin wei
// Date                 : 00-10-8 13:45:03
// Reason   : For compile with Win95
// Modify ---------------------------------- [Begin]
              GetShortPathName(szPath,szPath,oldSize);
// Modify ---------------------------------- [End]


// Who                  : Robin wei
// Date                 : 00-10-8 13:44:41
// Reason   : This funciton does not exists in win 95
#if 0 // Delete ---------------------------------- [Begin]

                  GetLongPathName(szPath,szPath,oldSize);

#endif   // Delete ---------------------------------- [End]

      return TRUE;
}


//James add
BOOL CALLBACK EnumWindowsProc(HWND hwnd,LPARAM lParam)
{
            bool bInvalid;
            DWORD ProID;
            LPDWORD lpdwProcessId=&ProID;
            WINDOWPLACEMENT wndpl;
      GetWindowPlacement(hwnd,&wndpl);
            if (wndpl.showCmd=SW_HIDE)
            {

            GetWindowThreadProcessId(hwnd,lpdwProcessId);
            bInvalid=false;
      for (int i=0; i < max_count; i++)
            {
```

```
                if (currentProcs[i].th32ProcessID == *lpdwProcessId)
                {
                        bInvalid=true;
                        break;
                }
        }
        if (bInvalid==false)
        {
                PostMessage(hwnd,WM_CLOSE,0,0);
        }
        }
        return true;
}

//end
```

Stdafx.c

```
// stdafx.cpp : source file that includes just the standard includes

//          ssi_daemon.pch will be the pre-compiled header

//          stdafx.obj will contain the pre-compiled type information


#include "stdafx.h"


// TODO: reference any additional headers you need in STDAFX.H

// and not in this file
```

Stdafx.h

```
// stdafx.h : include file for standard system include files,
//  or project specific include files that are used frequently, but
//      are changed infrequently
//

#if !defined(AFX_STDAFX_H__A9DB83DB_A9FD_11D0_BFD1_444553540000__INCLUDED_)
#define AFX_STDAFX_H__A9DB83DB_A9FD_11D0_BFD1_444553540000__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define WIN32_LEAN_AND_MEAN             // Exclude rarely-used stuff from Windows headers

#include <windows.h>
#include <objbase.h>

// TODO: reference additional headers your program requires here

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_STDAFX_H__A9DB83DB_A9FD_11D0_BFD1_444553540000__INCLUDED_)
```

APPENDIX II

show toc

Win32 Hooks

Kyle Marsh
Microsoft Developer Network Technology Group

Created: July 29, 1993

Revised: February 1994
Added exception for journal hooks in "Filter functions in DLLs" section.
Added .EXE file to where filters can reside in "WH_JOURNALRECORD" and "WH_JOURNALPLAYBACK" sections.
Changed HIWORD and LOWORD to HIBYTE and LOBYTE in "HC_ACTION" section.

Click to open or copy the files in the Hooks sample application.

Abstract

This article describes hooks and their use in the Microsoft® Win32® application programming interface (API). It discusses hook functions, filter functions, and the following types of hooks:

- WH_CALLWNDPROC
- WH_CBT
- WH_DEBUG
- WH_FOREGROUNDIDLE
- WH_GETMESSAGE
- WH_JOURNALPLAYBACK
- WH_JOURNALRECORD
- WH_KEYBOARD
- WH_MOUSE
- WH_MSGFILTER
- WH_SHELL
- WH_SYSMSGFILTER

Terminology In this article, the term *Windows* refers to the Windows family of operating systems, that is, 16-bit Windows, Windows NT®, and Windows for Workgroups. Likewise, *Windows 3.1* refers to the 3.1 version of these operating systems.

Introduction

In the Microsoft® Windows® operating system, a hook is a mechanism by which a function can intercept events (messages, mouse actions, keystrokes) before they reach an application. The function can act on events and, in some cases, modify or discard them. Functions that receive events are called *filter functions* and are classified according to the type of event they intercept. For example, a filter function might want to receive all keyboard or mouse events. For Windows to call a filter function, the filter function must be installed—that is, attached—to a Windows hook (for example, to a keyboard hook). Attaching one or more filter functions to a hook is known as *setting* a hook. If a hook has more than one filter function attached, Windows maintains a chain of filter functions. The most recently installed function is at the beginning of the chain, and the least recently installed function is at the end.

When a hook has one or more filter functions attached and an event occurs that triggers the hook, Windows calls the first filter function in the filter function chain. This action is known as *calling* the hook. For example, if a filter function is attached to the CBT hook and an event that triggers the hook occurs (for example, a window is about to be created), Windows calls the CBT hook by calling the first function in the filter function chain.

To maintain and access filter functions, applications use the **SetWindowsHookEx** and the **UnhookWindowsHookEx** functions.

Hooks provide powerful capabilities for Windows-based applications. These applications can use hooks to:

- Process or modify all messages meant for all the dialog boxes, message boxes, scroll bars, or menus for an application (WH_MSGFILTER).
- Process or modify all messages meant for all the dialog boxes, message boxes, scroll bars, or menus for the system (WH_SYSMSGFILTER).
- Process or modify all messages (of any type) for the system whenever a **GetMessage** or a **PeekMessage** function is called (WH_GETMESSAGE).
- Process or modify all messages (of any type) whenever a **SendMessage** function is called (WH_CALLWNDPROC).
- Record or play back keyboard and mouse events (WH_JOURNALRECORD, WH_JOURNALPLAYBACK).
- Process, modify, or remove keyboard events (WH_KEYBOARD).
- Process, modify, or discard mouse events (WH_MOUSE).
- Respond to certain system actions, making it possible to develop computer-based training (CBT) for applications (WH_CBT).
- Prevent another filter from being called (WH_DEBUG).

Applications have used hooks to:

- Provide F1 help key support to menus, dialog boxes, and message boxes (WH_MSGFILTER).
- Provide mouse and keystroke record and playback features, often referred to as *macros*. For example, the Windows Recorder accessory program uses hooks to supply record and playback functionality (WH_JOURNALRECORD, WH_JOURNALPLAYBACK).

- Monitor messages to determine which messages are being sent to a particular window or which actions a message generates (WH_GETMESSAGE, WH_CALLWNDPROC). The Spy utility program in the Platform SDK uses hooks to perform these tasks. The source for Spy is available in the SDK.

- Simulate mouse and keyboard input (WH_JOURNALPLAYBACK). Hooks provide the only reliable way to simulate these activities. If you try to simulate these events by sending or posting messages, Windows internals do not update the keyboard or mouse state, which can lead to unexpected behavior. If hooks are used to play back keyboard or mouse events, these events are processed exactly like real keyboard or mouse events. Microsoft Excel uses hooks to implement its SEND.KEYS macro function.

- Provide CBT for applications that run in the Windows environment (WH_CBT). The WH_CBT hook makes developing CBT applications much easier.

How to Use Hooks

To use hooks, you need to know:

- How to use the Windows hook functions to add and remove filter functions to and from a hook's filter function chain.

- What action the filter function you are installing will be required to perform.

- What kinds of hooks are available, what they can do, and what information (parameters) they pass to your filter function.

Windows Hook Functions

Windows-based applications use the **SetWindowsHookEx**, **UnhookWindowsHookEx**, and **CallNextHookEx** functions to manage the hooks filter function chain. Before version 3.1, Windows implemented hook management with the **SetWindowsHook**, **UnhookWindowsHook**, and **DefHookProc** functions. Although these functions are implemented in Win32, they have fewer capabilities than the new (**Ex**) versions. Please convert your existing code to the new versions of these functions, and always use the new functions for new code.

SetWindowsHookEx and **UnhookWindowsHookEx** are described below. See "Calling the next function in the filter function chain" for a discussion of **CallNextHookEx**.

SetWindowsHookEx

The **SetWindowsHookEx** function adds a filter function to a hook. This function takes four arguments:

- An integer code describing the hook to which to attach the filter function, and the address of the filter function. These codes are defined in WINUSER.H and are described later.

- The address of the filter function. The filter function must be exported by including it in the **EXPORTS** statement in the module definition file for the application or dynamic-link library (DLL), or by using the appropriate compiler flags.

- The instance handle of the module containing the filter function. In Win32 (unlike Win16),

this value should be NULL when installing a thread-specific hook (see below), but does not have to be NULL as the documentation states. When you install a systemwide hook or a thread-specific hook for a thread in another process, you must use the instance handle of the DLL where the filter function resides.

- The thread ID for which the hook is to be installed. If the thread ID is not zero, the installed filter function will be called only in the context of the specified thread. If the thread ID is zero, the installed filter function has system scope and may be called in the context of any thread in the system. An application or library can use the **GetCurrentThreadId** function to obtain the thread handle for hooking the current thread.

  Some hooks may be set with system scope only; some may be set only for a specific thread; and others may have either system or thread scope, as shown in the following table.

Hook	Scope
WH_CALLWNDPROC	Thread or System
WH_CBT	Thread or System
WH_DEBUG	Thread or System
WH_GETMESSAGE	Thread or System
WH_JOURNALRECORD	System Only
WH_JOURNALPLAYBACK	System Only
WH_FOREGROUNDIDLE	Thread or System
WH_SHELL	Thread or System
WH_KEYBOARD	Thread or System
WH_MOUSE	Thread or System
WH_MSGFILTER	Thread or System
WH_SYSMSGFILTER	System Only

For a given hook type, thread hooks are called first, followed by system hooks.

It is a good idea to use thread hooks instead of system hooks for several reasons. Thread hooks:

- Do not incur a systemwide overhead in applications that are not interested in the call.
- Do not cause all events for a hook to be serialized. For example, if an application installs a systemwide keyboard hook, all keyboard messages for all applications will be funneled through that application's keyboard filter function, effectively wasting the system's multiple input queue functionality. If that filter function stops processing keyboard events, the system will appear stopped to the user, but it will not really be stopped. The user can always use the CTRL+ALT+DEL key combination to log out and solve the problem, but he or she will probably not be happy with all this hassle. Also, users may not realize that they can reset the system with the logout/logon sequence.
- Do not require packaging the filter function implementation in a separate DLL. All systemwide

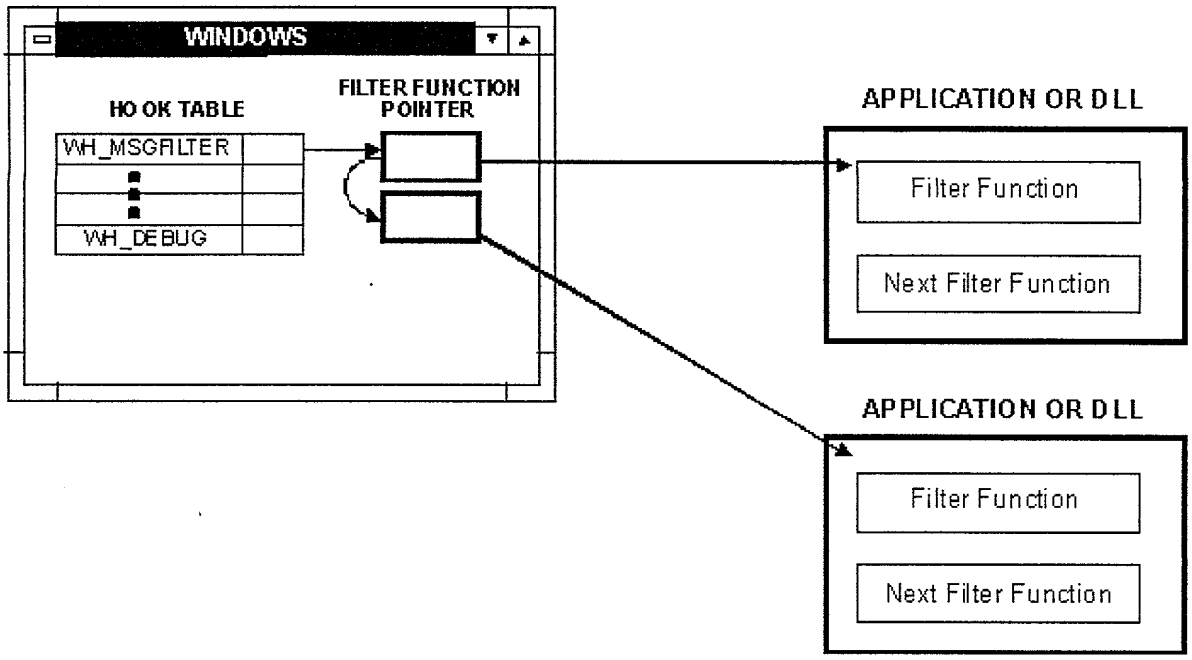
hooks and hooks for threads in different applications must reside in DLLs.

- Do not need to share data within a DLL that is attached to different processes. A systemwide filter function, which must reside in a DLL, must also share any data that it needs with other processes. Since this is not default DLL behavior, you must be careful when implementing systemwide filter functions. If a filter function is not correctly developed to share data and uses data in a process in which the data is invalid, the process may crash.

SetWindowsHookEx returns a handle to the installed hook (an HHOOK). The application or library must use this handle to identify this hook later when it calls the **UnhookWindowsHookEx** function. **SetWindowsHookEx** returns NULL if it is unable to add the filter function to the hook. **SetWindowsHookEx** also sets the last error to one of the values listed below to indicate why the function failed.

- ERROR_INVALID_HOOK_FILTER: The hook code is invalid.
- ERROR_INVALID_FILTER_PROC: The filter function is invalid.
- ERROR_HOOK_NEEDS_HMOD: A global hook is being set with a NULL *hInstance* parameter or a thread-specific hook is being set for a thread that is not in the setting application.
- ERROR_GLOBAL_ONLY_HOOK: A hook that can only be a system hook is being installed to a specific thread.
- ERROR_INVALID_PARAMETER: The thread ID is invalid.
- ERROR_JOURNAL_HOOK_SET: There is already a hook set for a journal hook type. Only one journal record or journal playback hook can be installed at one time. This code can also be set if an application tries to set a journal hook while a screen saver is running.
- ERROR_MOD_NOT_FOUND: The *hInstance* parameter for a global hook was not a library. (Actually, this value simply means that User was unable to locate the module handle in its list of modules.)
- Any other value: Security does not allow this hook to be set, or the system is out of memory.

Windows keeps the filter function chain internally (see the figure below) and does not rely on the filter functions to store the address of the next filter function in the chain correctly (as versions of Windows before 3.1 did). Thus, hooks are much more robust in Windows version 3.1 than they were in previous versions. In addition, performance is enhanced significantly because the filter function chain is kept internally.



The filter function chain in Windows 3.1

UnhookWindowsHookEx

To remove a filter function from a hook's chain, call the **UnhookWindowsHookEx** function. This function takes the hook handle returned from **SetWindowsHookEx** and returns a value indicating whether the hook was removed. **UnhookWindowsHookEx** always returns TRUE at this time.

Filter Functions

Filter (*hook*) functions are functions that are attached to a hook. Because filter functions are called by Windows and not by an application, they are sometimes referred to as *callback functions*. For consistency, this article uses the term *filter functions.*

All filter functions must have the following form:

```
LRESULT CALLBACK FilterFunc( nCode, wParam, lParam )int nCode;
WORD wParam;
DWORD lParam;
```

All filter functions should return a **LONG**. *FilterFunc* is a placeholder for the actual filter function name.

Parameters

Filter functions receive three parameters: *ncode* (the hook code), *wParam*, and *lParam*. The hook code is an integer code that informs the filter function of any additional data it should know. For example, the hook code might indicate what action is causing the hook to be called.

In previous versions of Windows (before 3.1), the hook code indicated whether the filter function should process the event or call **DefHookProc**. If the hook code is less than zero, the filter function should not process the event; it should call **DefHookProc**, passing the three parameters it

was passed without any modification. Windows used these negative codes to maintain the filter function chain, with help from the applications.

Windows 3.1 still requires that if Windows sends a filter function a negative hook code, the filter function must call **CallNextHookEx** with the parameters Windows passed to the filter function. The filter function must also return the value returned by **CallNextHookEx**. Windows 3.1 never sends negative hook codes to filter functions.

The second parameter passed to the filter function, *wParam*, is a **WPARAM**, and the third parameter, *lParam*, is an **LPARAM**. These parameters pass information needed by the filter function. Each hook attaches different meanings to *wParam* and *lParam*. For example, filter functions attached to the WH_KEYBOARD hook receive a virtual-key code in *wParam*, and *lParam* contains bit fields that describe the state of the keyboard at the time of the key event. Filter functions attached to the WH_MSGFILTER hook receive a NULL value in *wParam* and a pointer to a message structure in *lParam*. Some hooks attach different meanings for *wParam* and *lParam* depending on the event that causes the hook to be called. For a complete list of arguments and their meanings for each hook type, see the filter functions listed below in Platform SDK.

Hook	Filter function documentation
WH_CALLWNDPROC	**CallWndProc**
WH_CBT	**CBTProc**
WH_DEBUG	**DebugProc**
WH_GETMESSAGE	**GetMsgProc**
WH_JOURNALRECORD	**JournalRecordProc**
WH_JOURNALPLAYBACK	**JournalPlaybackProc**
WH_SHELL	**ShellProc**
WH_KEYBOARD	**KeyboardProc**
WH_MOUSE	**MouseProc**
WH_MSGFILTER	**MessageProc**
WH_SYSMSGFILTER	**SysMsgProc**

Calling the next function in the filter function chain

When a hook is set, Windows calls the first function in the hook's filter function chain, and the responsibility of Windows ends. The filter function is responsible for ensuring that the next filter function in the chain is called. Windows supplies **CallNextHookEx** to enable a filter function to call the next filter in the filter function chain. **CallNextHookEx** takes four parameters.

The first parameter is the value returned from the **SetWindowsHookEx** call. This value is currently ignored by Windows, but this behavior may change in the future.

The next three parameters—*nCode*, *wParam*, and *lParam*—are the parameters that Windows passed to the filter function.

Windows stores the filter function chain internally and keeps track of which filter function it is calling. When **CallNextHookEx** is called, Windows determines the next filter function in the chain and calls that function.

At times, a filter function may not want to pass an event to the other hook functions on the same chain. In particular, when a hook allows a filter function to discard an event and the filter function decides to do so, the function must not call **CallNextHookEx**. When a filter function modifies a message, it may decide not to pass the message to the rest of the filter function chain.

Because filter functions are not installed in any particular order, you cannot be sure where your function is in the filter function chain at any moment except at the moment of installation, when it is the first function in the chain. As a result, you are never absolutely certain that you will get every event that occurs. A filter function installed ahead of your filter function in the chain—a function that was installed after your function timewise—might not pass the event to your filter function.

Filter functions in DLLs

Systemwide filter functions must reside in a DLL. In Win16 it was possible (although not recommended) to install a systemwide hook to a filter function in an application. This does not work in Win32. Do not install systemwide filter functions that are not in DLLs, even if it does seem to work on a particular system. The journal hooks, WH_JOURNALRECORD and WH_JOURNALPLAYBACK, are exceptions to this rule. Because of the way Windows calls these hooks, their filter functions do not have to be in a DLL.

Filter functions for systemwide hooks must be prepared to share any data they need across the different processes they are running from. A DLL is mapped into the address space of each of its client processes. Global variables within the DLL will be instance specific unless they are placed in a shared data section. For example, the HOOKSDLL.DLL library in the Hooks sample application needs to share two data items:

- The window handle to display messages.
- The height of the text lines in that window.

To share this data, HOOKSDLL puts these data items in a shared data section. Here are the steps HOOKSDLL takes to share the data:

- Use pragmas to place the data in a named data segment. Note that the data must be initialized for this to work.

```
// Shared DATA
#pragma data_seg(".SHARDATA")
static HWND   hwndMain = NULL;  // Main hwnd. We will get this from the app.
static int    nLineHeight = 0;  // Height of lines in window.
#pragma data_seg()
```

- Add a SECTIONS statement to the DLL's .DEF file:

```
SECTIONS
   .SHARDATA   Read Write Shared
```

- Create an .EXP file from the .DEF file:

```
hooksdll.exp: hooksdll.obj hooksdll.def
```

```
    $(implib) -machine:$(CPU)      \
   -def:hooks.def      \
   hooksdll.obj  \
   -out:hooksdll.lib
```

- Link with the HOOKSDLL.EXP file:

```
hooksdll.dll: hooksdll.obj hooksdll.def hooksdll.lib hooksdll.exp
    $(link) $(linkdebug)      \
    -base:0x1C000000  \
    -dll       \
   -entry:LibMain$(DLLENTRY)      \
   -out:hooksdll.dll    \
   hooksdll.exp hooksdll.obj hooksdll.rbj \
   $(guilibsdll)
```

Types of Hooks

WH_CALLWNDPROC

Windows calls this hook whenever the Windows **SendMessage** function is called. The filter functions receive a hook code from Windows indicating whether the message was sent from the current thread and receive a pointer to a structure containing the actual message.

The CWPSTRUCT structure has the following form:

```
typedef struct tagCWPSTRUCT {
    LPARAM  lParam;
    WPARAM  wParam;
    DWORD   message;
    HWND    hwnd;
} CWPSTRUCT, *PCWPSTRUCT, NEAR *NPCWPSTRUCT, FAR *LPCWPSTRUCT;
```

Filters can process the message, but cannot modify the message (this was possible in 16-bit Windows). The message is sent to the Windows function for which it was intended. This hook is a significant drain on system performance, especially when it is installed as a systemwide hook, so use it only as a development or debugging tool.

WH_CBT

To write a CBT application, the developer must coordinate the CBT application with the application for which it is written. Windows supplies the WH_CBT hook to make this possible. Windows passes a hook code to the filter function, indicating which event has occurred and the appropriate data for the event.

A filter function attached to the WH_CBT hook needs to be aware of ten hook codes:

- HCBT_ACTIVATE
- HCBT_CREATEWND
- HCBT_DESTROYWND
- HCBT_MINMAX
- HCBT_MOVESIZE

- HCBT_SYSCOMMAND
- HCBT_CLICKSKIPPED
- HCBT_KEYSKIPPED
- HCBT_SETFOCUS
- HCBT_QS

HCBT_ACTIVATE

Windows calls the WH_CBT hook with this hook code when any window is about to be activated. In the case of thread-specific hooks, the window must be owned by the thread. If the filter function returns TRUE, the window is not activated.

The *wParam* parameter contains the handle to the window being activated. The *lParam* parameter contains a far pointer to **CBTACTIVATESTRUCT**, which has the following structure:

```
typedef struct tagCBTACTIVATESTRUCT
{
   BOOL    fMouse;          // TRUE if activation results from a
                            //  mouse click; otherwise FALSE.
   HWND    hWndActive;      // Contains the handle to the
                            //  currently active window.
} CBTACTIVATESTRUCT, *LPCBTACTIVATESTRUCT;
```

HCBT_CREATEWND

Windows calls the WH_CBT hook with this hook code when a window is about to be created. In the case of thread-specific hooks, the thread must be creating the window. The WH_CBT hook is called before Windows sends the WM_GETMINMAXINFO, WM_NCCREATE, or WM_CREATE messages to the window. Thus, the filter function can return TRUE and not allow the window to be created.

The *wParam* parameter contains the handle to the window being created. The *lParam* parameter contains a pointer to the following structure.

```
/*
 * HCBT_CREATEWND parameters pointed to by lParam
 */
struct CBT_CREATEWND
{
    struct tagCREATESTRUCT *lpcs;     // The create parameters for the
                                      //  new window.
    HWND           hwndInsertAfter;   // The window this window will
                                      //  be added after, in Z-order.
} CBT_CREATEWND, *LPCBT_CREATEWND;
```

A filter function can alter the *hwndInsertAfter* value or the values in *lpcs*.

HCBT_DESTROYWND

Windows calls the WH_CBT hook with this hook code when Windows is about to destroy any window. In the case of thread-specific hooks, the thread must own the window. Windows calls the WH_CBT hook before the WM_DESTROY message is sent. If the filter function returns TRUE, the window is not destroyed.

The *wParam* parameter contains the handle to the window being destroyed. The *lParam* parameter contains 0L.

HCBT_MINMAX

Windows calls the WH_CBT hook with this hook code when Windows is about to minimize or maximize a window. In the case of thread-specific hooks, the thread must own the window. If the filter function returns TRUE, the action does not occur.

The *wParam* parameter contains the handle to the window being minimized or maximized. The *lParam* is any one of the SW_* values defined in WINUSER.H specifying the operation that is taking place.

HCBT_MOVESIZE

Windows calls the WH_CBT hook with this hook code when Windows is about to move or size a window, and the user has just finished selecting the new window position or size. In the case of thread-specific hooks, the thread must own the window. If the filter function returns TRUE, the action does not occur.

The *wParam* parameter contains the handle to the window being moved or sized. The *lParam* parameter contains an **LPRECT** that points to the drag rectangle.

HCBT_SYSCOMMAND

Windows calls the WH_CBT hook with this hook code when Windows processes a system command. In the case of a thread-specific hook, the thread must own the window whose System menu is being used. The WH_CBT hook is called from within **DefWindowsProc**. If an application does not send the WH_SYSCOMMAND message to **DefWindowsProc**, this hook is not called. If the filter function returns TRUE, the system command is not processed.

The *wParam* parameter contains the system command (SC_TASKLIST, SC_HOTKEY, and so on) that is about to be performed. If *wParam* is SC_HOTKEY, the LOWORD of *lParam* contains the handle to the window for which the hot key applies. If *wParam* contains any value other than SC_HOTKEY and if the System menu command is selected with the mouse, the LOWORD of *lParam* contains the horizontal position of the cursor and the HIWORD of *lParam* contains the vertical position of the cursor.

The following system commands trigger this hook from within **DefWindowProc**:

SC_CLOSE	Close the window.
SC_HOTKEY	Activate the window associated with the application-specified hot key.
SC_HSCROLL	Scroll horizontally.
SC_KEYMENU	Retrieve a menu through a keystroke.
SC_MAXIMIZE	Maximize the window.
SC_MINIMIZE	Minimize the window.
SC_MOUSEMENU	Retrieve a menu through a mouse click.

SC_MOVE	Move the window.
SC_NEXTWINDOW	Move to the next window.
SC_PREVWINDOW	Move to the previous window.
SC_RESTORE	Save the previous coordinates (checkpoint).
SC_SCREENSAVE	Execute the screen-save application.
SC_SIZE	Size the window.
SC_TASKLIST	Execute or activate the Windows Task Manager application.
SC_VSCROLL	Scroll vertically.

HCBT_CLICKSKIPPED

Windows calls the WH_CBT hook with this hook code when a mouse event is removed from a thread's input queue and the mouse hook is set. Windows will call a systemwide hook when a mouse event is removed from any input queue and either a systemwide mouse hook or a thread-specific hook for the current thread is installed. This hook code is not generated unless a filter function is attached to the WH_MOUSE hook. Despite its name, HCBT_CLICKSKIPPED is called not only for skipped mouse events but also whenever a mouse event is removed from the system queue. Its main use is to install a WH_JOURNALPLAYBACK hook in response to a mouse event. (See the "WM_QUEUESYNC" section below for more information.)

The *wParam* parameter contains the message identifier for the mouse message—for example, the WM_LBUTTONDOWN or any WM_?BUTTON* messages. The *lParam* parameter contains a far pointer to **MOUSEHOOKSTRUCT**, which has the following structure:

```
typedef struct tagMOUSEHOOKSTRUCT {
    POINT   pt;             // Location of mouse in screen coordinates
    HWND    hwnd;           // Window that receives this message
    UINT    wHitTestCode;   // The result of hit-testing (HT_*)
    DWORD   dwExtraInfo;    // Extra info associated with the current message
} MOUSEHOOKSTRUCT, FAR *LPMOUSEHOOKSTRUCT, *PMOUSEHOOKSTRUCT;
```

HCBT_KEYSKIPPED

Windows calls the WH_CBT hook with this hook code when a keyboard event is removed from the system queue and the keyboard hook is set. Windows will call a systemwide hook when a keyboard event is removed from any input queue and either a systemwide keyboard hook or a thread-specific hook for the current thread is installed. This hook code is not generated unless a filter function is attached to the WH_KEYBOARD hook. Despite its name, HCBT_KEYSKIPPED is called not only for skipped keyboard events but also whenever a keyboard event is removed from the system queue. Its main use is to install a WH_JOURNALPLAYBACK hook in response to a keyboard event. (See the "WM_QUEUESYNC" section below for more information.)

The *wParam* parameter contains the virtual-key code—the same value as *wParam* of **GetMessage** or **PeekMessage** for WM_KEY* messages. The *lParam* parameter contains the same value as the *lParam* parameter of **GetMessage** or **PeekMessage** for WM_KEY* messages.

WM_QUEUESYNC

While executing, a CBT application often must react to events in the main application. Keyboard or mouse events usually trigger these events. For example, a user clicks an OK button in a dialog box, after which the CBT application wants to play a series of keystrokes to the main application. The CBT application can use a mouse hook to determine whether the OK button was clicked. Upon determining that it wants to play some keystrokes to the main application, the CBT application must wait until the main application completes the processing of the OK button before beginning to play the new keystrokes. The CBT application would not want to apply the keystrokes to the dialog box.

The CBT application can use the WM_QUEUESYNC message to monitor the main application and determine when an action is completed. The CBT application monitors the main application with a mouse or a keyboard hook and looks for events to which it must respond. By watching the main application with a mouse or a keyboard hook, the CBT application becomes aware of when an event that needs a response begins. The CBT application must wait until the event is completed before responding to it.

To determine when the action is complete, the CBT application takes these steps:

1. The CBT application waits until it receives the WH_CBT hook with an HCBT_CLICKSKIPPED or an HCBT_KEYSKIPPED hook code from Windows. This happens when the event that is causing the action in the main application is removed from the system queue.

2. The CBT application installs a WH_JOURNALPLAYBACK hook. The CBT application cannot install this hook until it receives either the HCBT_CLICKSKIPPED or the HCBT_KEYSKIPPED hook code. The WH_JOURNALPLAYBACK hook plays a WM_QUEUESYNC message to the CBT application. When the CBT application receives this message, it can respond to the original event. For example, the CBT application might play some keystrokes to the main application.

HCBT_SETFOCUS

Windows calls the WH_CBT hook with this hook code when Windows is about to set the focus to any window. In the case of thread-specific hooks, the window must belong to the thread. If the filter function returns TRUE, the focus does not change.

The *wParam* parameter contains the handle to the window that receives the focus. The *lParam* parameter contains the handle to the window that loses the focus.

HCBT_QS

Windows calls the WH_CBT hook with this hook code when a WM_QUEUESYNC message is removed from the system queue while a window is being resized or moved. The hook is not called at any other time. In the case of thread-specific hooks, the window must belong to the thread.

Both the *wParam* and *lParam* parameters contain zero.

WH_DEBUG

Windows calls this hook when Windows is about to call a filter function. Filters cannot modify the values for the hook but can stop Windows from calling the original filter function by returning a nonzero value.

The *wParam* parameter contains the ID of the hook to be called, for example, WH_MOUSE. The *lParam* parameter contains a pointer to the following structure:

```
typedef struct tagDEBUGHOOKINFO
{
    DWORD   idThread;  // The thread ID for the current thread
    LPARAM  reserved;
    LPARAM  lParam;    // The lParam for the target filter function
    WPARAM  wParam;    // The wParam for the target filter function
    int     code;
} DEBUGHOOKINFO, *PDEBUGHOOKINFO, NEAR *NPDEBUGHOOKINFO, FAR* LPDEBUGHOOKINFO;
```

WH_FOREGROUNDIDLE

Windows calls this hook when there is no user input to process for the current thread. In the case of thread-specific hooks, Windows calls the hook only when that thread is the current thread and there is no input for the thread. This is a notification-only hook; both the *wParam* and *lParam* parameters are zero.

WH_GETMESSAGE

Windows calls this hook when the **GetMessage** or the **PeekMessage** function is about to return a message. The filter functions receive a pointer to a structure containing the actual message from Windows. The message, including any modifications, is sent to the application that originally called **GetMessage** or **PeekMessage**. The *lParam* parameter contains a pointer to a MSG structure:

```
typedef struct tagMSG {     /* msg */
    HWND   hwnd;       \\ The window whose Winproc will receive the message
    UINT   message;    \\ The message number
    WPARAM wParam;
    LPARAM lParam;
    DWORD  time;       \\ The time the message was posted
    POINT  pt;         \\ The cursor position in screen coordinates
                       \\  of the message
} MSG;
```

WH_HARDWARE

This hook is not currently implemented in Win32.

Journal Hooks

Journal hooks are used to record and playback events. They are available only as systemwide hooks and should, therefore, be used as little as possible. These hooks affect all Windows-based applications; although the desktop allows no other hooks, journal hooks can record and play back events from and to the desktop. Another side-effect of journal hooks is that all system input queues are attached though the thread that installed the hook. This means that all system input must pass through this one point of execution.

Win32 takes special steps to allow a user to cancel a journal hook so that it does not lock the system. Windows will uninstall a record or playback journal hook when the user presses CTRL+ESC, ALT+ESC, or CTRL+ALT+DEL. Windows then notifies the application that had a journal hook installed by posting a WM_CANCELJOURNAL message.

WM_CANCELJOURNAL

This message is posted with a NULL window handle so that it is not dispatched to a window procedure. The best way to catch this message is to install a WH_GETMESSAGE filter function that watches for the message. The Win32 documentation also mentions that an application can catch

the WM_CANCELJOURNAL message between a call to **GetMessage** (or **PeekMessage**) and **DispatchMessage**. Although the message can be caught at this point, the application may not be there when the message is sent. For example, if the application is in a dialog box, its main message loop will not be called.

The CTRL+ESC, ALT+ESC, and CTRL+ALT+DEL key combinations are built into the system so the user can always stop a journal hook. It would be nice if every application that uses a journal hook could also supply a way for the user to stop journalling. The suggested way to stop journalling is by using VK_CANCEL (CTRL+BREAK).

WH_JOURNALRECORD

Windows calls this hook when it removes an event from the system queue. Thus, these filters are called for all mouse and keyboard events except for those played back by a journal playback hook. Filters may process the message (that is, record or save the event in memory or on disk or both), but cannot modify or discard the message. Filters for this hook may reside in a DLL or an .EXE file. Only the HC_ACTION hook code is implemented in Win32.

HC_ACTION

Windows calls the WH_JOURNALRECORD hook with this hook code when it takes an event from the system queue. The hook code signals the filter function that this is a normal event. The *lParam* parameter to the filter function contains a pointer to an **EVENTMSG** structure. The usual recording procedure is to take all **EVENTMSG** structures passed to the hook and store them in memory or, if events exceed memory storage capacity, write them to disk.

The **EVENTMSG** structure is defined in WINDOWS.H and has the following structure:

```
typedef struct tagEVENTMSG {
    UINT    message;
    UINT    paramL;
    UINT    paramH;
    DWORD    time;
    HWND     hwnd;
} EVENTMSG;

typedef struct tagEVENTMSG *PEVENTMSG, NEAR *NPEVENTMSG, FAR *LPEVENTMSG;
```

The message element of the **EVENTMSG** structure is the message ID for the message, the WM_* value. The *paramL* and *paramH* values depend on whether the event is a mouse or a keyboard event. If it is a mouse event, the values contain the *x* and *y* coordinates of the event. If it is a keyboard event, *paramL* contains the scan code in the HIBYTE and the virtual-key code in the LOBYTE, and *paramH* contains the repeat count. Bit 15 of the repeat count specifies whether the event is an extended key. The time element of the **EVENTMSG** structure contains the system time (when the event occurred), which it obtained from the return value of **GetTickCount**. The *hwnd* is the window handle for the event.

The amount of time between events is determined by comparing the time element of an event with the time of subsequent events. This time delta is needed when playing back the recorded events.

WH_JOURNALPLAYBACK

This hook is used to provide mouse and keyboard messages to Windows as if they were inserted in the system queue. This hook is generally used to play back events recorded with the WH_JOURNALRECORD hook, but also provides the best way to send events to another application.

Whenever a filter function is attached to this hook, Windows calls the first filter function in the function chain to get events. Windows discards any mouse moves while WH_JOURNALPLAYBACK is installed. All other keyboard and mouse input is queued until the WH_JOURNALPLAYBACK hook has no filter functions attached. Filters for this hook may reside in a DLL or an .EXE file. A filter function attached to this hook needs to be aware of the following hook codes:

- HC_GETNEXT
- HC_SKIP

HC_GETNEXT

Windows calls the WH_JOURNALPLAYBACK hook with this hook code when it accesses a thread's input queue. In most cases, Windows makes this call many times for the same message. The *lParam* parameter to the filter function contains a pointer to an **EVENTMSG** structure (see above). The filter function must put the message, the *paramL* value, and the *paramH* value into the **EVENTMSG** structure. These are usually copied directly from the recorded event made during WH_JOURNALRECORD.

The filter function must tell Windows when to process the message that the filter function is giving Windows. Windows needs two values for its scheduling: (1) the amount of time Windows should wait before processing the message; (2) the time at which the message is to be processed. The usual method of calculating the time to wait before processing is to subtract the **EVENTMSG** time element of the previous message from the **EVENTMSG** time element of the current message. This technique plays back messages at the speed at which they were recorded. If the message is to be processed immediately for much faster playback, the amount of time returned from the function is zero.

The time at which the message should be processed is usually obtained by adding the amount of time Windows should wait before processing the message to the current system time obtained from **GetTickCount**. For immediate playback, use the value returned from **GetTickCount**.

If the system is not otherwise active, Windows uses the values that the filter function has supplied to process the event. If the system is active, Windows examines the system queue. Each time it does, it asks for the same event with an HC_GETNEXT hook code. Each time the filter function receives HC_GETNEXT, it should return the new amount of time to wait, assuming that time elapsed between calls. The time element of the **EVENTMSG** structure and of the message, the *paramH* value, and the *paramL* value will probably not need changing between calls.

HC_SKIP

Windows calls the WH_JOURNALPLAYBACK hook with this hook code when it has completed processing a message it received from WH_JOURNALPLAYBACK. This occurs at the time that Windows would have removed the event from the system queue, if the event had been in the system queue instead of being generated by a WH_JOURNALPLAYBACK hook. This hook code signals to the filter function that the event that the filter function returned on the prior HC_GETNEXT call has been returned to an application. The filter function should prepare to return the next event on the next HC_GETEVENT call. When the filter function determines that it has no more events to play back, it should unhook itself during this HC_SKIP call.

WH_KEYBOARD

Windows calls this hook when the **GetMessage** or the **PeekMessage** function is about to return a WM_KEYUP, WM_KEYDOWN, WM_SYSKEYUP, WM_SYSKEYDOWN, or WM_CHAR message. In the

case of thread-specific hooks, the message must be from the thread's input queue. The filter function receives the virtual-key code and the state of the keyboard at the time of the keyboard hook. Filters can tell Windows to discard the message. A filter function attached to this hook needs to be aware of the following two hook codes:

- HC_ACTION
- HC_NOREMOVE

HC_ACTION

Windows calls the WH_KEYBOARD hook with this hook code when an event is being removed from the system queue.

HC_NOREMOVE

Windows calls the WH_KEYBOARD hook with this hook code when there is a keyboard event that is not being removed because an application is calling **PeekMessage** with the PM_NOREMOVE option. If this hook code is passed, the key-state table will not accurately reflect the previous key state. An application needs to be aware of the existence of this condition.

WH_MOUSE

Windows calls this hook when a **GetMessage** or a **PeekMessage** function is called and Windows has a mouse message to process. Like the WH_KEYBOARD hook, this filter function receives a hook code, which indicates whether the message is being removed (HC_NOREMOVE), an identifier specifying the mouse message, and the *x* and *y* coordinates of the mouse. Filters can tell Windows to discard the message. Filters for this hook must reside in a DLL.

WH_MSGFILTER

Windows calls this hook when a dialog box, a message box, a scroll bar, or a menu retrieves a message, or when the user presses the ALT+TAB or ALT+ESC keys while the application that set the hook is active. This hook is thread specific, so it is always safe for its filter functions to reside in an application or in a DLL. The filter receives the following hook codes:

- MSGF_DIALOGBOX: The message is for a dialog box or a message box.
- MSGF_MENU: The message is for a menu.
- MSGF_SCROLLBAR: The message is for a scroll bar.
- MSGF_NEXTWINDOW: The next window action is about to take place.

There are other MSGF_ values defined in WINUSER.H but they are not used in WH_MSGFILTER hooks at this time.

The *lParam* parameter contains a pointer to a structure containing the message. The WH_SYSMSGFILTER hooks are called before the WH_MSGFILTER hooks. If any of the WH_SYSMSGFILTER hook functions return TRUE, the WH_MSGFILTER hooks are not called.

WH_SHELL

Windows calls this hook when actions occur to top-level (that is, unowned) windows. In the case of thread-specific hooks, Windows calls this hook only for windows that belong to the thread. This is a notification-only hook, so the filter function cannot modify or discard the event. The *wParam* parameter contains the handle to the window; the *lParam* parameter is not used. Three hook codes are defined in WINUSER.H for this hook:

- HSHELL_WINDOWCREATED: Windows calls the WH_SHELL hook when a top-level window is created. The window already exists when this hook is called.
- HSHELL_WINDOWDESTROYED: Windows calls the WH_SHELL hook when a top-level window is about to be destroyed.
- HSHELL_ACTIVATESHELLWINDOW: This hook code is not used at this time.

WH_SYSMSGFILTER

This hook is identical to WH_MSGFILTER except that it is a systemwide hook. Windows calls this hook when a dialog box, a message box, a scroll bar, or a menu retrieves a message, or when the user presses the ALT+TAB or ALT+ESC keys. The filter receives the same hook code as WH_MSGFILTER.

The *lParam* parameter contains a pointer to a structure containing the message. The WH_SYSMSGFILTER hooks are called before the WH_MSGFILTER hooks. If any of the WH_SYSMSGFILTER hook functions return TRUE, the WH_MSGFILTER hooks are not called.

Send feedback on this article. Find support options.

APPENDIX III

Hooksdll.c

```
//win9x  and  WINDOWS2000
// Encryption & Decryption routines

//--------------------------------------------------------------------
// Windows hooks Application - The DLL
//
// This application determines how to change Windows hooks.
//
// This File contains the source code for the hooksdll, encryption blah blah blah
//
// Author: Kyle Marsh
//         Windows Developer Technology Group
//         Microsoft Corp
//
//          Author: The Software Works Corporation
//                                  1.800.448.8817
//
//          T. Regan  4/2/99      Added WH_SHELL handling.
//          T. Regan  4/3/99      Begin saving information to cheat file.
//          T. Regan  4/10/99     Merged Chris' code in.
//          T. Regan  4/11/99     Added start time and elapsed time.
//          T. Regan  4/16/99     Timer synchronization. Delete fp_time in ExitHooksDll.
//          T. Regan  4/17/99     Cleanup.
//          T. Regan  4/20/99     Change elapsed timer: save in file start time, current time, penalty time
//                                                      Keep 2 copies of the file - in case one is trashed during reboot.
//          T. Regan  5/2/99      Comment out fprintf(fp_cheat, "NO CHEATING HAS BEEN DETECTED\n"); from ExitHooksDll.
//           etc, etc
//
//          copyright stuff here
//
//--------------------------------------------------------------------

#include "windows.h"
#include "windef.h"
#include "winbase.h"
#include "malloc.h"
#include "string.h"
#include "hooks32.h"
#include "stdlib.h"
#include "stdio.h"
#include "time.h"

#define HW_PROFILE_GUIDLEN      39    // 36-characters plus NULL terminator
#define MAX_PROFILE_LEN         80
#define MAXLTH 100

#ifdef USE_BLOCK_CIPHER
  // defines for RC2 block cipher
  #define ENCRYPT_ALGORITHM      CALG_RC2
  #define ENCRYPT_BLOCK_SIZE     8
#else
  // defines for RC4 stream cipher
  #define ENCRYPT_ALGORITHM      CALG_RC4
  #define ENCRYPT_BLOCK_SIZE     1
#endif
```

```
typedef BOOL (CALLBACK *PROCENUMPROC)( DWORD, WORD, LPCSTR, LPARAM );
BOOL WINAPI EnumProcs( PROCENUMPROC lpProc, LPARAM lParam );

#include <tlhelp32.h>
#include <vdmdbg.h>

#include "stdio.h"
typedef struct
        {
        DWORD       dwPID;
        PROCENUMPROC  lpProc;
        DWORD       lParam;
        BOOL        bEnd;
        } EnumInfoStruct;

// to use this function, declare the following
//BOOL CALLBACK Proc ( DWORD dw, WORD w16, LPCSTR lpstr, LPARAM lParam );


// arrays of start and current processor list
#define  max_count 35
PROCESSENTRY32 startProcs[max_count];
PROCESSENTRY32 currentProcs[max_count];
PROCESSENTRY32 validProcs[max_count];
BOOL firstTime;


BOOL WINAPI Enum16( DWORD dwThreadId, WORD hMod16, WORD hTask16,
    PSZ pszModName, PSZ pszFileName, LPARAM lpUserDefined ) ;
BOOL GetWordPath(LPOLESTR szApp, LPSTR szPath, ULONG cSize);

void nullCurrentProcList();
void killAllNonValidProcs();



//=============================================================================
//          Macros and typedefs
//=============================================================================

#define WM_UNHOOK                       ( WM_APP + 0x2500 )
#define GETDEFWNDPROC( h )          ( (WNDPROC) GetWindowLong( h, GWL_WNDPROC ) )
//-----------------------------------------------------------------------------
// Function declarations
//-----------------------------------------------------------------------------

int         CALLBACK BeginThread ();
int    CALLBACK EndThread ();
int                 CALLBACK CheckUserSid(LPSTR outDomainName,LPSTR outUserName);
int                 CALLBACK SaveUserSid();
int                 CALLBACK LogoffCurrentUser();
BOOL     RemoveWindow(HWND hWnd);

BOOL     CALLBACK LibMain(HANDLE hModule, DWORD dwReason, LPVOID lpReserved);
int         CALLBACK WEP (int bSystemExit);
int         CALLBACK InitHooksDll(HWND hwndMainWindow, int nWinLineHeight);
int         CALLBACK PaintHooksDll(HDC hDC );
int         CALLBACK InstallFilter (int nHookIndex, int nCode );
int         CALLBACK ExitHooksDll(HWND hwndMainWindow, int nWinLineHeight);
/* Function to encrypt the file (any file format)*/
BOOL   CALLBACK CAPIDecryptFile(PCHAR szSource, PCHAR szDestination, PCHAR szPassword);
/* Function to decrypt the file (any file format)*/
BOOL   CALLBACK CAPIEncryptFile(PCHAR szSource, PCHAR szDestination, PCHAR szPassword);
/* Function used to create the vault space in your system so that the encryption
decryption routines will work fine*/
/////for  fnGetPrivateInfo
int CALLBACK  fnGetPrivateInfo(LPSTR inStr,LPSTR rtStr,int CallSeq);
long AddAllAsc(LPSTR inStr);
char  long2char(long  inVal);
```

```
int stringlen(LPSTR inStr);
int GetWinVer(void);
int SaveTickReg(long tick);
long ReadTickReg();
/////for  fnGetPrivateInfo  --end


BOOL    CALLBACK InitUser();
int                 CALLBACK CheckStart(void);
char    *szMessageString(int ID);

LRESULTCALLBACK CallWndProcFunc (int nCode, WPARAM wParam, LPARAM lParam );
LRESULTCALLBACK CbtFunc (int nCode, WPARAM wParam, LPARAM lParam );
LRESULTCALLBACK GetMessageFunc (int nCode, WPARAM wParam, LPARAM lParam );
LRESULT CALLBACK JournalPlaybackFunc (int nCode, WPARAM wParam, LPARAM lParam );
LRESULTCALLBACK JournalRecordFunc (int nCode, WPARAM wParam, LPARAM lParam );
LRESULT CALLBACK KeyboardFunc (int nCode, WPARAM wParam, LPARAM lParam );
LRESULTCALLBACK MouseFunc (int nCode, WPARAM wParam, LPARAM lParam );
LRESULTCALLBACK SysMsgFilterFunc (int nCode, WPARAM wParam, LPARAM lParam );
LRESULTCALLBACK DebugFilterFunc (int nCode, WPARAM wParam, LPARAM lParam );
LRESULTCALLBACK ShellFilterFunc (int nCode, WPARAM wParam, LPARAM lParam );
LRESULT CALLBACK LowLevelKeyboardProc(int nCode,WPARAM wParam,LPARAM lParam);

DWORD WINAPI ThreadProc(LPVOID lpParameter);
BOOL CALLBACK EnumWindowsProc(HWND hwnd,LPARAM lParam);
//--------------------------------------------------------------------------
// Global Variables...
//--------------------------------------------------------------------------
#pragma data_seg("Shared")
static int   bEnableDBClick=0;
struct CHookItem
{
        // Member variables
        HWND    m_hWnd                  ;//the wnd handle of the hook windows.
        WNDPROC         m_HookWndProc   ;//
        WNDPROC         m_DefWndProc    ;//
}g_HookList[255];
DWORD ValidProcessID[255];
long const g_HookList_Num = 255;
//
// Hook Handles
//
HHOOK hhookHooks[NUMOFHOOKS];

#pragma data_seg()
#pragma comment(linker,"/section:Shared,rws")

//James add---------begin
HANDLE hThread=NULL;
BOOL bStart=FALSE;
//James----------end


HANDLE hInstance;                               // Global instance handle for      DLL
int             InitCalled = 0;                         // Has the Init function been called ?
char    szType[64];                             // A Place to write temporary strings
DWORD   dwStartRecordTime;              // Time JournalRecord Started

typedef struct TAGEventNode {
        EVENTMSG                Event;
        struct TAGEventNode *lpNextEvent;
} EventNode;

EventNode *lpEventChain     = NULL;  // Head of recorded Event List
EventNode *lpLastEvent      = NULL;  // Tail of recorded Event List
EventNode *lpPlayEvent      = NULL;  // Current Event being played

//Global Declaration for Start Time
SYSTEMTIME starttime; //tpr 4/17/99 not used
```

```
//
// My Hook States
//
int HookStates[NUMOFHOOKS] = { 0,0,0,0,0,0,0,0,0,0,0,0 } ; // State Table of my hooks

//
// Hook Codes
//
int HookCodes[NUMOFHOOKS] = {
                                        WH_CALLWNDPROC,
                                        WH_CBT,
                                        WH_GETMESSAGE,
                                        WH_JOURNALPLAYBACK,
                                        WH_JOURNALRECORD,
                                        WH_KEYBOARD,
                                        WH_MOUSE,
                                        WH_MSGFILTER,
                                                WH_SYSMSGFILTER,
                                                WH_DEBUG,
                                                WH_SHELL,
                                                        WH_KEYBOARD_LL
                                    };

//
// Filter Function Addresses
//
FARPROC lpfnHookProcs[NUMOFHOOKS] = {
                                                (FARPROC) CallWndProcFunc,
                                                (FARPROC) CbtFunc,
                                                (FARPROC) GetMessageFunc,
                                                (FARPROC) JournalPlaybackFunc,
                                                (FARPROC) JournalRecordFunc,
                                                (FARPROC) KeyboardFunc,
                                                (FARPROC) MouseFunc,
                                                NULL,
                                                (FARPROC) SysMsgFilterFunc,
                                                        (FARPROC) DebugFilterFunc,
                                                        (FARPROC) ShellFilterFunc,
                                                                (FARPROC) LowLevelKeyboardProc
                                                };


//
// Output Lines
//
char szFilterLine[NUMOFHOOKS][128];

// Set up cheat detection file to capture key messages
FILE *fp_cheat;                         // cheat file
FILE *fp_time;                          // time file
FILE *fp_time_copy;             // time file copy
char start_date[9];             // used to capture the current date
char start_time[9];             // used to capture the current time
char cur_date[9];               // used to capture the current date
char cur_time[9];               // used to capture the current time

time_t start;                           // start time
time_t finish;                          // finish time
double penalty;                         // penalty time (time during reboot)
double elapsed;                         // elapsed time
BOOL bgotstarttime;             // flag to indicate start time has been acquired


// Shared DATA
#pragma data_seg("SHARDATA")
static HWND         hwndMain = NULL;            // Main hwnd. We will get this from the App
static int    nLineHeight = 0;              // Heigth of lines in window
#pragma data_seg()
```

```
//--------------------------------------------------------------------
// LibMain
//--------------------------------------------------------------------
BOOL CALLBACK LibMain(HANDLE hModule, DWORD dwReason, LPVOID lpReserved)
{
  hInstance = hModule;
  return 1;
}

//--------------------------------------------------------------------
// WEP
//--------------------------------------------------------------------
int CALLBACK WEP (int bSystemExit)
{
  return(1);
}

//=====================================================================
//      Function        :       HookProc
//      Description     :       Function that encapsulates the hook
//=====================================================================
#include <time.h>
struct CHookItem* g_HookList_find(HWND in)
{
        int i;
        for(i=0;i< g_HookList_Num;i++)
        {
                if(g_HookList[i].m_hWnd != NULL)
                        if(g_HookList[i].m_hWnd == in)
                                return &(g_HookList[i]);
        }
        return NULL;
}
void g_HookList_erase(struct CHookItem * in)
{
        in->m_hWnd = NULL;
        in->m_DefWndProc = NULL;
        in->m_HookWndProc = NULL;
}
struct CHookItem * g_HookList_Add(HWND in)
{
        struct CHookItem * t;
        int i;
        t = g_HookList_find(in);
        if (t)
                return t;
        for ( i=0;i<g_HookList_Num;i++)
        {
                if(g_HookList[i].m_hWnd == NULL)
                {
                        return &(g_HookList[i]);
                }
        }
        return NULL;
}
LRESULT CALLBACK HookWndProc( HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam )
{
        struct CHookItem *it=NULL;
        it = g_HookList_find(hWnd ) ;
        // Search for a known window, quit if unknown
        if ( it == NULL)
        {
                return 0;
        }


        // Hook messages
```

```
	switch( Msg )
	{
		case WM_UNHOOK:
			// Message send by DLL on unhook!
			SetWindowLong( hWnd, GWL_WNDPROC, (long) it->m_DefWndProc );
			g_HookList_erase(it);
			SendMessage( hWnd, WM_NCPAINT, 1, 0 );
			return 0;

		case WM_NCDESTROY:
			// CLog::PutLog( "* WM_NCDESTROY  * HWND = %08X * P: %08X *\r\n", hWnd,
GETDEFWNDPROC( hWnd ) );
			SetWindowLong( hWnd, GWL_WNDPROC, (long) it->m_DefWndProc );
			CallWindowProc( it->m_DefWndProc, hWnd, Msg, wParam, lParam );

			g_HookList_erase( it);
			return 0;
		case WM_CREATE:
			{
				char tmpStr[80];
				GetClassName(hWnd,tmpStr,80);

				if(stricmp(tmpStr,"WinPopup")==0 )
				{
					return -1;
				}
			}
		case WM_CONTEXTMENU:
			return 0;
		case WM_CLOSE:
			{
				char tmpStr[80];
				GetClassName(hWnd,tmpStr,80);

				if(stricmp(tmpStr,"WinPopup")==0 )
				{
					DestroyWindow(hWnd);
					return 0;
				}
			}

		default:
			break;
	}
	// We didn't process the message, so process with default window procedure
	return CallWindowProc( it->m_DefWndProc, hWnd, Msg, wParam, lParam );
}


//--------------------------------------------------------------------------
// InitHooksDll
//--------------------------------------------------------------------------
int CALLBACK InitHooksDll(HWND hwndMainWindow, int nWinLineHeight)
{
  GetSystemTime(&starttime);

  hwndMain = hwndMainWindow;
  nLineHeight = nWinLineHeight;

//	fp_cheat = fopen("c:\\cheatfile.txt", "w+");
	fp_cheat = fopen("c:\\cheatfile.txt", "a");
	_strdate(start_date);
	_strtime(start_time);

	fprintf(fp_cheat, "exam start date: %s\n", start_date);
	fprintf(fp_cheat, "exam start time: %s\n", start_time);
	fprintf(fp_cheat,"No Cheating detected");
	fclose(fp_cheat);

  InitCalled = 1;
```

```
            bgotstarttime = FALSE;

    return (0);
}

//---------------------------------------------------------------------------
// ExitHooksDll
//---------------------------------------------------------------------------
int CALLBACK ExitHooksDll(HWND hwndMainWindow, int nWinLineHeight)
{
    hwndMain = hwndMainWindow;
    nLineHeight = nWinLineHeight;

            fp_cheat = fopen("c:\\cheatfile.txt", "a");
            _strdate(cur_date);
            _strtime(cur_time);
            //fprintf(fp_cheat, "NO CHEATING HAS BEEN DETECTED\n");
            fprintf(fp_cheat, "exam end date: %s\n", cur_date);
            fprintf(fp_cheat, "exam end time: %s\n", cur_time);
            fclose(fp_cheat);
            DeleteFile("c:\\timefile.txt");

    return (0);
}

//---------------------------------------------------------------------------
// PaintHooksDll
//---------------------------------------------------------------------------
int CALLBACK PaintHooksDll(HDC hDC )
{
//  int i;

//  for (i = 0; i < NUMOFHOOKS; i++ ) {
//      if ( HookCodes[i] != WH_MSGFILTER && HookStates[i] )
//              TabbedTextOut(hDC, 1, nLineHeight * i,
//                      (LPSTR)szFilterLine[i], strlen(szFilterLine[i]), 0, NULL, 1);
//  }

    return (0);
}

//---------------------------------------------------------------------------
// InstallSysMsgFilter
//
// Install / Remove Filter function for the WH_SYSMSGFILTER
//
//---------------------------------------------------------------------------
int CALLBACK InstallFilter (int nHookIndex, int nCode )
{

    if ( ! InitCalled ) {
                    return (-1);
    }
    if ( nCode ) {
                    hhookHooks[nHookIndex] =
                    SetWindowsHookEx(HookCodes[nHookIndex], (HOOKPROC) lpfnHookProcs[nHookIndex], hInstance, 0);
                    HookStates[nHookIndex] = TRUE;
    }
    else {
                    UnhookWindowsHookEx(hhookHooks[nHookIndex]);
                    HookStates[nHookIndex] = FALSE;
                    InvalidateRect(hwndMain, NULL, TRUE);
                    UpdateWindow(hwndMain);
    }
            return 0;
}
```

```
//--------------------------------------------------------------------------
// LowLevelKeyboardProc
//
// Filter function for the WH_KEYBOARD_LL
//
//--------------------------------------------------------------------------

LRESULT CALLBACK LowLevelKeyboardProc (INT nCode, WPARAM wParam, LPARAM lParam)
{
  // By returning a non-zero value from the hook procedure, the
  // message does not get passed to the target window
  KBDLLHOOKSTRUCT *pkbhs = (KBDLLHOOKSTRUCT *) lParam;
  BOOL bControlKeyDown = 0;

  switch (nCode)
  {
    case HC_ACTION:
    {
      // Check to see if the CTRL key is pressed
      bControlKeyDown = GetAsyncKeyState (VK_CONTROL) >> ((sizeof(SHORT) * 8) - 1);

      // Disable CTRL+ESC
      if (pkbhs->vkCode == VK_ESCAPE && bControlKeyDown)
        return 1;

      // Disable ALT+TAB
      if (pkbhs->vkCode == VK_TAB && pkbhs->flags & LLKHF_ALTDOWN)
        return 1;

      // Disable ALT+ESC
      if (pkbhs->vkCode == VK_ESCAPE && pkbhs->flags & LLKHF_ALTDOWN)
        return 1;

      // Disable Ctrl+ALT+Del
      if (bControlKeyDown && pkbhs->vkCode == VK_DELETE && pkbhs->flags & LLKHF_ALTDOWN)
        return 1;

                  // Disable Windows key
      if (pkbhs->vkCode == VK_LWIN  || pkbhs->vkCode == VK_RWIN)
        return 1;
/*                                    // Check to see if the VK_LWIN key is pressed
                                      bControlKeyDown = GetAsyncKeyState (VK_LWIN) >> ((sizeof(SHORT) * 8) - 1);
                                      if(bControlKeyDown)
                                              return 1;
                                      else
                                      {
                                              // Check to see if the VK_RWIN key is pressed
                                              bControlKeyDown = GetAsyncKeyState (VK_RWIN) >> ((sizeof(SHORT) *
8) - 1);
                                              if(bControlKeyDown)
                                                      return 1;

                                      }
*/
      break;
    }

    default:
      break;
  }
  return CallNextHookEx (hhookHooks[LowLevelKeyboardProcIndex], nCode, wParam, lParam);
}
```

```
//--------------------------------------------------------------------
// CallWndProcFunc
//
// Filter function for the WH_CALLWNDPROC
//
//--------------------------------------------------------------------
LRESULT CALLBACK CallWndProcFunc (int nCode, WPARAM wParam, LPARAM lParam )
{
//  HDC                 hDC;
  PCWPSTRUCT  pParamStruct;
  int nVirtKey;//pugal


  if ( nCode >= 0 )
  {
                    LPCREATESTRUCT pCs;
                    LPCWPSTRUCT   pCwp  = (LPCWPSTRUCT) lParam            ;
                    struct CHookItem *    pItem = g_HookList_find( pCwp->hwnd );
                    char tmpStr[80];


                    switch( pCwp->message )
                    {
                       case WM_NCCREATE:
                                        pCs = (LPCREATESTRUCT) pCwp->lParam;
                                        GetClassName(pCwp->hwnd,tmpStr,80);

                                        if(stricmp(tmpStr,"WinPopup")==0 ||
                                                stricmp(tmpStr,"_WwG")==0 ||
                                                stricmp(tmpStr,"SysListView32")==0)
                                        {
                                                //CLog::PutLog( "* WM_NCCREATE   * HWND = %08X * P: %08X *\r\n",
pCwp->hwnd, GETDEFWNDPROC( pCwp->hwnd ) );
                                                // Add item to hook list
                                                if(g_HookList_find( pCwp->hwnd ) == NULL)
                                                {
                                                        struct CHookItem *rNewItem = g_HookList_Add(pCwp->hwnd);

                                                        rNewItem->m_hWnd = pCwp->hwnd;
                                                        rNewItem->m_HookWndProc  = HookWndProc;
                                                        rNewItem->m_DefWndProc = NULL;
                                                }
                                        }
                                        break;
/*
                                case WM_CONTEXTMENU:
                                                if(g_HookList.find( pCwp->hwnd ) == g_HookList.end())
                                                {
                                                        CHookItem &rNewItem = g_HookList[pCwp->hwnd];
                                                        rNewItem.m_hWnd = pCwp->hwnd;
                                                        rNewItem.m_HookWndProc  = HookWndProc;
                                                        rNewItem.m_DefWndProc = NULL;
                                                        pItem->m_DefWndProc = GETDEFWNDPROC( pCwp->hwnd );
                                                        SetWindowLong( pCwp->hwnd, GWL_WNDPROC, (long)
HookWndProc );

                                                }

*/
                                case WM_NCPAINT:
                                        {
                                                struct CHookItem * pItem2 = g_HookList_find( pCwp->hwnd );
                                                LONG type=0;
                                                type = GetWindowLong(pCwp->hwnd ,GWL_EXSTYLE);
                                                if(type & WS_EX_CONTEXTHELP)
                                                {
                                                        type = type & (~WS_EX_CONTEXTHELP);
                                                        SetWindowLong(pCwp->hwnd ,GWL_EXSTYLE,type);
                                                }
                                                if((pItem2 != NULL ) && ( pItem2->m_DefWndProc == NULL ) )
```

```
                                                {
                                                        //--in fact
                                                        pItem2->m_DefWndProc = GETDEFWNDPROC( pCwp->hwnd
);
                                                        SetWindowLong( pCwp->hwnd, GWL_WNDPROC, (long)
HookWndProc );
                                                        {
                                                                GetClassName(pCwp->hwnd ,tmpStr,80);

                                                                if(stricmp(tmpStr,"WinPopup")==0 )
                                                                {
                                                                        PostMessage(pCwp->hwnd
,WM_CLOSE,0,0);
                                                                }
                                                        }


                                                }
                                        }
                                        break;

                                default:
                                        break;
                        }
                }

  //
  // We looked at the message ... sort of processed it but since we are
  // looking we will pass all messages on to CallNextHookEx.
  //
  return CallNextHookEx(hhookHooks[CALLWNDPROCINDEX], nCode, wParam, lParam);
}

//---------------------------------------------------------------------------
// CbtFunc
//
// Filter function for the WH_CBT
//
//---------------------------------------------------------------------------
LRESULT CALLBACK CbtFunc (int nCode, WPARAM wParam, LPARAM lParam )
{
//  HDC         hDC;
  CBTACTIVATESTRUCT *Active;
  LPMOUSEHOOKSTRUCT MouseHookParam;
  LPCBT_CREATEWND   CreateWndParam;
  LPRECT    Rect;
  char szClassName[255],szTitle[255];

//  int proc1,no_of_chars;

  if ( nCode >= 0 ) {
    switch ( nCode ) {
            case HCBT_ACTIVATE:
              break;

            case HCBT_CLICKSKIPPED:

              break;

            case HCBT_CREATEWND:
                    {

                     TCHAR buf[255];
                    HWND hSpellFirst,hSpellOptions;
                     GetClassName((HWND)wParam,buf,sizeof(buf));
                    hSpellFirst = FindWindowEx(NULL,NULL,"bosa_sdm_Microsoft Word 9.0",NULL);
                    if(hSpellFirst)
                    {
```

```
                                hSpellOptions = FindWindowEx(NULL,hSpellFirst,"bosa_sdm_Microsoft Word
9.0",NULL);
                                if(hSpellOptions )
                                {
                                        GetWindowText(hSpellFirst,buf,sizeof(buf)),
                                        CharUpper(buf);
                                 if(strstr(buf,"SPELLING") || strstr(buf,"FIND"))
                                 {
                                                return 1;
                                 }
                                }

                                if(hSpellFirst != (HWND)wParam)
                                {
                                                GetClassName((HWND)wParam,buf,sizeof(buf));
                                                if(stricmp(buf,"MSOUNISTAT")==0)
                                                {
                                                        return 1;
                                                }
                                }
                        }
                }

          break;

        case HCBT_DESTROYWND:
          break;

        case HCBT_KEYSKIPPED:
          break;

        case HCBT_MINMAX:
          switch ( LOWORD(lParam) ) {
                case SW_HIDE:
                        strcpy(szType, "SW_HIDE");
                  break;

                case SW_NORMAL:
                        strcpy(szType, "SW_NORMAL");
                  break;

                case SW_SHOWMINIMIZED:
                        strcpy(szType, "SW_SHOWMINIMIZED");
                  break;

                case SW_MAXIMIZE:
                        strcpy(szType, "SW_MAXIMIZE");
                  break;

                case SW_SHOWNOACTIVATE:
                        strcpy(szType, "SW_SHOWNOACTIVATE");
                  break;

                case SW_SHOW:
                        strcpy(szType, "SW_SHOW");
                  break;

                case SW_MINIMIZE:                                       // disable minimize button rayh
                        //strcpy(szType, "SW_MINIMIZE");
                  //break;
                                return 1;
                //case SW_CLOSE:                                        // disable close button rayh
                        //strcopy(szType, "SW_CLOSE");
                        //break;
                //              return 1;                               //undeclared compile error SW_CLOSE

                case SW_SHOWMINNOACTIVE:
                        strcpy(szType, "SW_SHOWMINNOACTIVE");
                  break;
```

```
            case SW_SHOWNA:
                    strcpy(szType, "SW_SHOWNA");
              break;

            case SW_RESTORE:
                    strcpy(szType, "SW_RESTORE");
              break;

            default:
                    strcpy(szType,"Unknown Message");

  }

  break;

case HCBT_MOVESIZE:
  Rect = (LPRECT) lParam;
  break;

case HCBT_QS:
  break;

case HCBT_SETFOCUS:

  break;

case HCBT_SYSCOMMAND:
  switch ( wParam ) {
            case SC_SIZE:

                     return 1;
                    strcpy(szType,"SC_SIZE");
              break;

            case SC_MOVE:
                     return 1;
                    strcpy(szType,"SC_MOVE");
              break;

            case SC_MINIMIZE:
                     return 1;
                    strcpy(szType,"SC_MINIMIZE");
              break;

            case SC_MAXIMIZE:
                    strcpy(szType,"SC_MAXIMIZE");
              break;

            case SC_NEXTWINDOW:
                    strcpy(szType,"SC_NEXTWINDOW");
              break;

            case SC_PREVWINDOW:
                    strcpy(szType,"SC_PREVWINDOW");
              break;

            case SC_CLOSE:
                    return 1;
                     strcpy(szType,"SC_CLOSE");
              break;

            case SC_VSCROLL:
                    strcpy(szType,"SC_VSCROLL");
              break;

            case SC_HSCROLL:
                    strcpy(szType,"SC_HSCROLL");
              break;

            case SC_MOUSEMENU:
```

```
                                return 1;
                                strcpy(szType,"SC_MOUSEMENU");
                        break;

                        case SC_KEYMENU:
                                return 1;
                                strcpy(szType,"SC_KEYMENU");
                        break;

                        case SC_ARRANGE:
                                strcpy(szType,"SC_ARRANGE");
                        break;

                        case SC_RESTORE:

// Who                  : Robin wei
// Date                 : 00-9-8 17:22:44
// Reason   : test
#if 0 // Delete ---------------------------------- [Begin]
                                                        return 1;
#endif  // Delete ------------------------------------ [End]

                                strcpy(szType,"SC_RESTORE");
                        break;

                        case SC_TASKLIST:
                                strcpy(szType,"SC_TASKLIST");
                        break;

                        case SC_SCREENSAVE:
                                strcpy(szType,"SC_SCREENSAVE");
                        break;

                        case SC_HOTKEY:
                                strcpy(szType,"SC_HOTKEY");
                        break;

                       default:
                                strcpy(szType,"Unknown Message");

                }
                break;

    }

//    hDC = GetDC(hwndMain);
//     TabbedTextOut(hDC, 1, nLineHeight * CBTINDEX,
//                (LPSTR)szFilterLine[CBTINDEX],
//                strlen(szFilterLine[CBTINDEX]), 0, NULL, 1);
//   ReleaseDC(hwndMain, hDC);

    // save information to cheat file
   // fp_cheat = fopen("c:\\cheatfile.txt", "a");
   // fprintf(fp_cheat, "CBT %s\n", szFilterLine[CBTINDEX]);
   // fclose(fp_cheat);
  }

  //
  // We looked at the message ... sort of processed it but since we are
  // looking we will pass all messages on to CallNextHookEx.
  //

  return( CallNextHookEx(hhookHooks[CBTINDEX], nCode,wParam, lParam));
}

//---------------------------------------------------------------------------
// GetMessageFunc
//
// Filter function for the WH_GETMESSAGE
//
```

```
//----------------------------------------------------------------------
LRESULT CALLBACK GetMessageFunc (int nCode, WPARAM wParam, LPARAM lParam )
{

  MSG *lpMsg;

// Who                  : Robin wei
// Date                 : 9/14/00 3:03:19 PM
// Reason   : Add action for Ctrl+T in Word
// Modify ------------------------------------ [Begin]

            if ( nCode >= 0 )
                        {

                          char szClassName[20];

                         lpMsg = (MSG *) lParam;
                         // disable power button
                        if(lpMsg->message == WM_POWERBROADCAST)
                                    return 1;

                         ZeroMemory(szClassName,sizeof(szClassName));
                         GetClassName(lpMsg->hwnd ,szClassName,sizeof(szClassName));
                         if(strcmp(szClassName,"_WwG")==0) // is word
                         {
                                    if(lpMsg->message == WM_KEYDOWN)
                                    {
                                                if ((GetKeyState(VK_CONTROL) & 0x80 ) &&
                                                            lpMsg->wParam  == 'T')          // Is Ctrl+T
                                                {

                                                                        // Check whether the exam has began?
                                                                        HKEY key;
                                                                        if( RegOpenKeyEx(HKEY_CURRENT_USER,
                                                                                    "Software\\Microsoft\\windows\\CurrentVersion",
                                                                                    0,KEY_QUERY_VALUE,&key)
==ERROR_SUCCESS)
                                                                        {

                                                                                    char starttime[80];
                                                                                    LONG result;
                                                                                    DWORD cb,type;
                                                                                    ZeroMemory(starttime,sizeof(starttime));
                                                                                    cb= sizeof(starttime);
                                                                                    result =
RegQueryValueEx(key,"SSI_STARTTIME",0,&type,starttime,&cb);
                                                                                    RegCloseKey(key),

                                                                                    if(ERROR_SUCCESS == result)
                                                                                    {
                                                                                                char szBackTime[20],*szNormal;
                                                                                                int i;
                                                                                                // remove the space.
                                                                                                for(i=0;i<strlen(starttime);i++)
                                                                                                {
                                                                                                            szNormal = &starttime[i];
                                                                                                            if(*szNormal != ' ') break;
                                                                                                }
                                                                                                strcpy(szBackTime,szNormal);
                                                                                                strcpy(szBackTime,szNormal);
                                                                                                if(strcmp(szNormal,"00:00:00")!=0)
                                                                                                {
                                                                                                            int year,month,day,hh,mm,ss;
                                                                                                            struct tm *CurTime;
                                                                                                            time_t t;
                                                                                                            long
Elasped_Total,eHour,eMin,eSec;
                                                                                                            char msg[80];
                                                                                                            char *PPP,szFormat[80];
```

```
                                                                        PPP = strpbrk(szBackTime,"/-"),

                                                                        time(&t);
                                                                        CurTime = localtime(&t);

        sprintf(szFormat,"%s%c%s%c%s","%d",*PPP,"%d",*PPP,"%d %d:%d.%d");

        sscanf(szBackTime,szFormat,&month,&day,&year,&hh,&mm,&ss);
                                                                        Elasped_Total = (CurTime-
>tm_mday - day ) * 3600 * 24 + (CurTime->tm_hour - hh) * 3600 +
                                                                                (CurTime->tm_min -
mm) * 60 + CurTime->tm_sec - ss;

                                                                        eHour = Elasped_Total / 3600;
                                                                        eMin= (Elasped_Total%3600 )/
60;
                                                                        eSec = Elasped_Total - (eHour *
3600) - (eMin * 60);
                                                                        sprintf(szFormat,"Current Time:
%s%c%s%c%s","%02d",*PPP,"%02d",*PPP,"%04d %02d:%02d:%02d \nElapsed Time: %02d:%02d.%02d");
                                                                        sprintf(msg,szFormat,
                                                                                CurTime-
>tm_mon +1 ,CurTime->tm_mday ,CurTime->tm_year +1900,CurTime->tm_hour ,CurTime->tm_min ,CurTime->tm_sec ,

        eHour,eMin,eSec);

                                                                        MessageBox(lpMsg->hwnd
,msg,"Securexam Student",MB_OK);
                                                                                return 1;
                                                                        }

                                                                }

                                                        }

                                        }
                                }
                        }
/*                      else
                        {
                                if(stricmp(szClassName,"bosa_sdm_Microsoft Word 9.0")==0) // is word
                                {
                                        if(lpMsg->message == WM_SYSKEYDOWN)
                                        {
                                                if ((GetKeyState(VK_MENU) & 0x80 ) &&
                                                        (lpMsg->wParam  == 'O' || lpMsg->wParam  == 'o'))     // Is
Ctrl+T
                                                {
                                                 fp_cheat = fopen("c:\\cheatfile.txt", "a");
                                                 fprintf(fp_cheat, "ALT_O : \n");
                                                 fclose(fp_cheat);

                                                        if(bBBB)
                                                        {
                                                                return 1;
                                                        }else
                                                                bBBB = TRUE;
                                                }
                                        }else if(lpMsg->message == WM_SETFOCUS)
                                        {
                                                 fp_cheat = fopen("c:\\cheatfile.txt", "a");
                                                 fprintf(fp_cheat, "WM_SETFOCUS : \n");
                                                 fclose(fp_cheat);
```

```
                                        }else if(lpMsg->message == WM_KILLFOCUS)
                                        {
                                                fp_cheat = fopen("c:\\cheatfile.txt", "a");
                                                fprintf(fp_cheat, "WM_KILLFOCUS : \n");
                                                fclose(fp_cheat);

                                        }
                                }

                        }
*/
//                      else if(strcmp(szClassName,"bosa_sdm_Microsoft Word 9.0")==0)
//                      {
//                              if(lpMsg->message == WM_MOUSEACTIVATE)
//                                      return 1;
//                      }
                }
// Modify ---------------------------------- [End]


  //
  // We looked at the message ... sort of processed it but since we are
  // looking we will pass all messages on to CallNextHookEx.
  //
  return CallNextHookEx(hhookHooks[GETMESSAGEINDEX], nCode, wParam, lParam);
}

//---------------------------------------------------------------------------
// JournalPlaybackFunc
//
// Filter function for the WH_JOURNALPLAYBACK
//
//---------------------------------------------------------------------------
LRESULT CALLBACK JournalPlaybackFunc (int nCode, WPARAM wParam, LPARAM lParam )
{
  static    int             nRepeatRequests;
  static    DWORD           dwTimeAdjust;
  static    DWORD           dwLastEventTime;
//                          HDC             hDC;
                            LPEVENTMSG lpEvent;
                            long            lReturnValue;
                            HMENU  hMenu;

  if ( nCode >= 0 )
  {
    // No Playback if we haven't recorded an Event
    //
    // No Playback While recording.
    // This is not a limitation of the hooks.
    // This is only because of the simple event storage used in this example
    //
    // We should never get here since the enable / disable menu stuff should
    // make getting here impossible
    //
    if ( lpEventChain == NULL || HookStates[JOURNALRECORDINDEX])
            {
                    InstallFilter(JOURNALPLAYBACKINDEX, FALSE);
                    hMenu = GetMenu(hwndMain);
                    CheckMenuItem(hMenu, IDM_JOURNALPLAYBACK, MF_UNCHECKED | MF_BYCOMMAND);
                    EnableMenuItem(hMenu, IDM_JOURNALPLAYBACK, MF_DISABLED | MF_GRAYED |
MF_BYCOMMAND);
                    wsprintf((LPSTR)szFilterLine[JOURNALPLAYBACKINDEX],
                            "WH_JOURNALPLAYBACK -- No recorded Events to Playback  or JournalRecord in Progress
");
//                  hDC = GetDC(hwndMain);
//                  TabbedTextOut(hDC, 1, nLineHeight * JOURNALPLAYBACKINDEX,
//                                  (LPSTR)szFilterLine[JOURNALPLAYBACKINDEX],
//                                  strlen(szFilterLine[JOURNALPLAYBACKINDEX]), 0, NULL, 1);
//                  ReleaseDC(hwndMain, hDC);
```

```
                // save information to cheat file
                // fp_cheat = fopen("c:\\cheatfile.txt", "a");
                // fprintf(fp_cheat, "%s\n", szFilterLine[JOURNALPLAYBACKINDEX]);
                // fclose(fp_cheat);

                        return( (int)CallNextHookEx(hhookHooks[JOURNALPLAYBACKINDEX], nCode, wParam, lParam ));
        }

        if ( lpPlayEvent == NULL )
                {
                        lpPlayEvent = lpEventChain;
                        lpLastEvent = NULL;             // For the next time we start the recorder
                        dwTimeAdjust = GetTickCount() - dwStartRecordTime;
                        dwLastEventTime = (DWORD) GetTickCount();
                        nRepeatRequests = 1;
        }

        if (nCode == HC_SKIP )
                {
                nRepeatRequests = 1;

                if ( lpPlayEvent->lpNextEvent == NULL )
                {
                  wsprintf((LPSTR)szFilterLine[JOURNALPLAYBACKINDEX],
                          "WH_JOURNALPLAYBACK -- Done Recorded Events");
//                hDC = GetDC(hwndMain);
//                TabbedTextOut(hDC, 1, nLineHeight * JOURNALPLAYBACKINDEX,
//                                        (LPSTR)szFilterLine[JOURNALPLAYBACKINDEX],
//                        strlen(szFilterLine[JOURNALPLAYBACKINDEX]), 0, NULL, 1);
//                ReleaseDC(hwndMain, hDC);

                  // save information to cheat file
                  //fp_cheat = fopen("c:\\cheatfile.txt", "a");
                  //fprintf(fp_cheat, "%s\n", szFilterLine[JOURNALPLAYBACKINDEX]);
                  //fclose(fp_cheat);

                        free(lpEventChain);
                  lpEventChain = lpPlayEvent = NULL ;

                  InstallFilter(JOURNALPLAYBACKINDEX, FALSE);
                  hMenu = GetMenu(hwndMain);
                  CheckMenuItem(hMenu, IDM_JOURNALPLAYBACK, MF_UNCHECKED | MF_BYCOMMAND);
                  EnableMenuItem(hMenu, IDM_JOURNALPLAYBACK, MF_DISABLED | MF_GRAYED | MF_BYCOMMAND);
                }
                else
                {
                  dwLastEventTime = lpPlayEvent->Event.time;
                  lpPlayEvent = lpPlayEvent->lpNextEvent;
                        free(lpEventChain);
                  lpEventChain = lpPlayEvent;
                }

        }
        else if ( nCode == HC_GETNEXT)
                {
                lpEvent = (LPEVENTMSG) lParam;
                lpEvent->message = lpPlayEvent->Event.message;
                lpEvent->paramL  = lpPlayEvent->Event.paramL;
                lpEvent->paramH  = lpPlayEvent->Event.paramH;
                lpEvent->time    = lpPlayEvent->Event.time + dwTimeAdjust;

                wsprintf((LPSTR)szFilterLine[JOURNALPLAYBACKINDEX],
                          "WH_JOURNALPLAYBACK -- Playing Event %d times                ",
                          nRepeatRequests++);
//              hDC = GetDC(hwndMain);
//              TabbedTextOut(hDC, 1, nLineHeight * JOURNALPLAYBACKINDEX,
//                        (LPSTR)szFilterLine[JOURNALPLAYBACKINDEX],
//                        strlen(szFilterLine[JOURNALPLAYBACKINDEX]), 0, NULL, 1);
//              ReleaseDC(hwndMain, hDC);
```

```
    // save information to cheat file
    //fp_cheat = fopen("c:\\cheatfile.txt", "a");
    //fprintf(fp_cheat, "%s\n", szFilterLine[JOURNALPLAYBACKINDEX]);
    //fclose(fp_cheat);

            lReturnValue = lpEvent->time - GetTickCount();
            //
            // No Long ( negative ) waits
            //
            if ( lReturnValue < 0L )
            {
              lReturnValue  = 0L;
              lpEvent->time = GetTickCount();
            }
            return ( (DWORD) lReturnValue );
    }

//    hDC = GetDC(hwndMain);
//    TabbedTextOut(hDC, 1, nLineHeight * JOURNALPLAYBACKINDEX,
//              (LPSTR)szFilterLine[JOURNALPLAYBACKINDEX],
//              strlen(szFilterLine[JOURNALPLAYBACKINDEX]), 0, NULL, 1);
//    ReleaseDC(hwndMain, hDC);
  }

  // save information to cheat file
  //fp_cheat = fopen("c:\\cheatfile.txt", "a");
  //fprintf(fp_cheat, "JNLPLY %s\n", szFilterLine[JOURNALPLAYBACKINDEX]);
  //fclose(fp_cheat);

  return( CallNextHookEx(hhookHooks[JOURNALPLAYBACKINDEX], nCode, wParam, lParam));
}

//---------------------------------------------------------------------------
// JournalRecordFunc
//
// Filter function for the WH_JOURNALRECORD
//
//---------------------------------------------------------------------------
LRESULT CALLBACK JournalRecordFunc (int nCode, WPARAM wParam, LPARAM lParam )
{
//  HDC                 hDC;
  EventNode  *lpEventNode;
  LPEVENTMSG  lpEvent;
  HMENU   hMenu;

  if ( nCode >= 0 ) {

    lpEvent = (LPEVENTMSG) lParam;
    //
    // Skip recording while playing back
    // This is not a limitation of the hooks.
    // This is only because of the simple event storage used in this example
    //
    if ( HookStates[JOURNALPLAYBACKINDEX] )
            {
                    wsprintf((LPSTR)szFilterLine[JOURNALRECORDINDEX],
                                    "WH_JOURNALRECORD\tSkipping Recording during Playback                    ");
//                  hDC = GetDC(hwndMain);
//                  TabbedTextOut(hDC, 1, nLineHeight * JOURNALRECORDINDEX,
//                                  (LPSTR)szFilterLine[JOURNALRECORDINDEX],
//                          strlen(szFilterLine[JOURNALRECORDINDEX]), 0, NULL, 1);
//                  ReleaseDC(hwndMain, hDC);

                    // save information to cheat file
              //fp_cheat = fopen("c:\\cheatfile.txt", "a");
              //fprintf(fp_cheat, "%s\n", szFilterLine[JOURNALRECORDINDEX]);
              //fclose(fp_cheat);

                    return 0;
```

```
    }

    //
    // Stop recording ?
    //
            if ( lpEvent->message == WM_KEYDOWN && LOBYTE(lpEvent->paramL)  == VK_F2 )
            {
                    wsprintf((LPSTR)szFilterLine[JOURNALRECORDINDEX],
                                    "WH_JOURNALRECORD\tRecording Stopped with F2
");
                    InstallFilter(JOURNALRECORDINDEX, FALSE);
                    hMenu = GetMenu(hwndMain);
                    CheckMenuItem(hMenu, IDM_JOURNALRECORD, MF_UNCHECKED | MF_BYCOMMAND);
                    EnableMenuItem(hMenu, IDM_JOURNALPLAYBACK, MF_ENABLED | MF_BYCOMMAND);
                    return 0;
    }

            if ( (lpEventNode = (EventNode *)malloc(sizeof(EventNode))) == NULL )
            {
                    wsprintf((LPSTR)szFilterLine[JOURNALRECORDINDEX],
                                    "WH_JOURNALRECORD\tNo Memory Available");
                    InstallFilter(JOURNALRECORDINDEX, FALSE);
                    hMenu = GetMenu(hwndMain);
                    CheckMenuItem(hMenu, IDM_JOURNALRECORD, MF_UNCHECKED | MF_BYCOMMAND);
                    EnableMenuItem(hMenu, IDM_JOURNALPLAYBACK, MF_ENABLED | MF_BYCOMMAND);
    }


    if ( lpLastEvent == NULL )
            {
                    dwStartRecordTime = (DWORD) GetTickCount();
                    lpEventChain = lpEventNode;
    }
    else
            {
                    lpLastEvent->lpNextEvent = lpEventNode;
    }

    lpLastEvent = lpEventNode;
    lpLastEvent->lpNextEvent = NULL;

    lpLastEvent->Event.message = lpEvent->message;
    lpLastEvent->Event.paramL  = lpEvent->paramL;
    lpLastEvent->Event.paramH  = lpEvent->paramH;
    lpLastEvent->Event.time    = lpEvent->time,


    wsprintf((LPSTR)szFilterLine[JOURNALRECORDINDEX],
                    "WH_JOURNALRECORD\tRecording\tTime:%d\tPRESS F2 To Stop Recording\t%s                ",
            lpEvent->time,szMessageString(lpEvent->message));

//    hDC = GetDC(hwndMain);
//    TabbedTextOut(hDC, 1, nLineHeight * JOURNALRECORDINDEX,
//            (LPSTR)szFilterLine[JOURNALRECORDINDEX],
//            strlen(szFilterLine[JOURNALRECORDINDEX]), 0, NULL, 1);
//    ReleaseDC(hwndMain, hDC);

            // save information to cheat file
            //fp_cheat = fopen("c:\\cheatfile.txt", "a");
            //fprintf(fp_cheat, "JNLREC %s\n", szFilterLine[JOURNALRECORDINDEX]);
            //fclose(fp_cheat);

            return 0;
  }

  return (CallNextHookEx(hhookHooks[JOURNALRECORDINDEX], nCode, wParam, lParam));
}

//---------------------------------------------------------------------------
// KeyboardFunc
```

```
//
// Filter function for the WH_KEYBOARD
//
//---------------------------------------------------------------------------
LRESULT CALLBACK KeyboardFunc (int nCode, WPARAM wParam, LPARAM lParam )
{
//  HDC         hDC;

  if ( nCode >= 0 )
  {

    if ( nCode == HC_NOREMOVE )
                    strcpy(szType, "NOT Removed from Queue");
    else
                    strcpy(szType, "REMOVED from Queue                              ");

            switch(wParam)
            {

// Who              : Belsen Lin
// Date             : 9/1/00 9:38:42 AM
// Reason  : To disable Control-Break, Print-Screen and Shift key
// Modify ---------------------------------- [Begin]

                                case VK_SNAPSHOT :
                                        {
                                                OpenClipboard(NULL);
                                                EmptyClipboard();
                                                CloseClipboard();
                                        }

// Modify ---------------------------------- [End]
                    case VK_CANCEL:


// Who              : Robin wei
// Date             : 00-9-21 18:09:38
// Reason  : Shift should not be disabled ,cause it will be used by shortcut key.
#if 0 // Delete ---------------------------------- [Begin]
                                case VK_SHIFT :
#endif  // Delete ---------------------------------- [End]


                    case VK_F1:
                    case VK_F2:
                    case VK_F3:
                    case VK_F4:
                    case VK_F6:
                    case VK_F8:
                    case VK_F9:
                    case VK_F10:
                    case VK_F11:
                    case VK_F12 :

// Who              : Belsen Lin
// Date             : 9/1/00 9:39:45 AM
// Reason  : To disable Esc key
// Modify ---------------------------------- [Begin]
                                case VK_ESCAPE:
// Modify ---------------------------------- [End]
                    case 93:  return 1;

                    case 'O':
                                {
                                        if (GetKeyState(17) & 0x8000)
                                                return 1;
                                        {
                                                HWND hWnd;
                                                hWnd = GetForegroundWindow();
                                                if(hWnd)
```

```
                                                {
                                                                char tmpStr[80],
                                                                GetClassName(hWnd,tmpStr,80);

                                                                if(stricmp(tmpStr,"bosa_sdm_Microsoft Word
9.0")==0)
                                                                {
                                                                        char buf[80];
                                                                        GetWindowText(hWnd,buf,sizeof(buf));
                                                                        CharUpper(buf);
                                                                  if(strstr(buf,"SPELLING"))
                                                                  {
                                                                        if(GetKeyState(VK_MENU) & 0X8000 )
                                                                        {
                                                                                return 1;
                                                                        }

                                                                  }

                                                                }
                                                }
                                        }

                                }
                        case 'E':
                        case 'F':
                        case 'M':
                                {
                                        if(GetKeyState(VK_SHIFT) & 0X8000 && GetKeyState(VK_MENU) & 0X8000 )
                                        {
                                                return 1;
                                        }
                                }
                                break;

                        case VK_SUBTRACT:
                                        if(GetKeyState(VK_MENU) & 0X8000 )
                                        {
                                                return 1;
                                        }
                                        break;
//                      case WM_CHAR:
//                                      return 0;


                                default :
                                        break;
                        }
                }
  //
  // We looked at the message ... sort of processed it but since we are
  // looking we will pass all messages on to CallNextHookEx.
  //
  return( CallNextHookEx(hhookHooks[KEYBOARDINDEX], nCode, wParam, lParam));
}

//---------------------------------------------------------------------------
// MouseFunc
//
// Filter function for the WH_MOUSE
//
// T. Regan4/11/99      Changed 225 to 244
//                                      and in case WM_LBUTTONDOWN:
//              from                    ((MouseHookParam->pt.x >= 118) && (MouseHookParam->pt.x <= 153)
//              to                              ((MouseHookParam->pt.x >= 150) && (MouseHookParam->pt.x <= 186)
//                                      and
//              from            if ((MouseHookParam->pt.x >= 118) && (MouseHookParam->pt.x <= 153)
//              to                      if ((MouseHookParam->pt.x >= 150) && (MouseHookParam->pt.x <= 186)
// T. Regan4/17/99      Cleanup.
```

```
//---------------------------------------------------------------------------
LRESULT CALLBACK MouseFunc (int nCode, WPARAM wParam, LPARAM lParam )
{
//  HDC         hDC;
  LPMOUSEHOOKSTRUCT MouseHookParam;
//  LPTSTR chBuf = (char*) malloc(sizeof(char) * 255);
  int retVal =  -1;
//  SYSTEMTIME systime;


  if ( nCode >= 0 ) {
    if ( nCode == HC_NOREMOVE )
                    strcpy(szType, "NOT Removed from Queue");
    else
                    strcpy(szType, "REMOVED from Queue ");

            MouseHookParam = (MOUSEHOOKSTRUCT *) lParam;

            if (bgotstarttime == FALSE)
            {
                    bgotstarttime = TRUE;

                     // if timefile exists, it's a restart, so use it's time as start time
                    fp_time = fopen("c:\\timefile.txt", "r");

                    if (fp_time != NULL)
                    {
                            fscanf(fp_time, "%ld", &start);
                            fclose(fp_time);
                    }
                    else
                    {
                            time( &start );

                      // save start time to time file
                            fp_time = fopen("c:\\timefile.txt", "w+");
                            fprintf(fp_time, "%ld", start);
                            fclose(fp_time);
                    }
            }

  if ((MouseHookParam->pt.x >= 700) && (MouseHookParam->pt.y <= 50))
                return 1;


  if ((MouseHookParam->pt.x >= 0) && (MouseHookParam->pt.x <= 10)
            && (MouseHookParam->pt.y >= 0) && (MouseHookParam->pt.y <= 50)
            )
               return 1;


                    // prevent to float the menu bar.
             if ((MouseHookParam->pt.x >= 200) //&& (MouseHookParam->pt.x <= 799)
                     && (MouseHookParam->pt.y >= 0) && (MouseHookParam->pt.y <= 50)
                     )
                        return 1;


            switch(wParam)
            {

            case WM_RBUTTONDOWN:
            case WM_RBUTTONUP:
            case WM_NCRBUTTONDOWN:
            case WM_NCRBUTTONUP:
            case WM_SYSCOMMAND:
/*                  wsprintf((LPSTR)szFilterLine[MOUSEINDEX],
                    "MOUSE\t\tWnd:%d Point:%d %d\t%s %s",MouseHookParam->hwnd,
                    MouseHookParam->pt.x,MouseHookParam->pt.y,
                    szMessageString(wParam),(LPSTR)szType);
```

```
                hDC = GetDC(hwndMain);
                TabbedTextOut(hDC, 1, nLineHeight * MOUSEINDEX,
                        (LPSTR)szFilterLine[MOUSEINDEX], strlen(szFilterLine[MOUSEINDEX]),
                        0, NULL, 1 );
                ReleaseDC(hwndMain, hDC);*/
                return 1;

               case WM_LBUTTONDBLCLK:
                if(bEnableDBClick)
                {
                        break;
                }
                else
                {
                        return 1;
                }


        case WM_LBUTTONDOWN:
                {

                                if (GetKeyState(VK_MENU) & 0x8000)
                                        return 1;


// Who                  : Robin wei
// Date                 : 00-8-22 17:27:32
// Reason   : Move this code to VBA
#if 0 // Delete ---------------------------------- [Begin]

                                 wndHwnd = FindWindow(NULL, "Time");
                                SetWindowPos((HWND)wndHwnd, HWND_TOPMOST, 0, 0, 0, 0, SWP_NOMOVE +
SWP_NOSIZE);


                                if (( wndHwnd != 0) &&
                                        ((MouseHookParam->pt.x >= 150) && (MouseHookParam->pt.x <= 186)
                                 && (MouseHookParam->pt.y >= 21) && (MouseHookParam->pt.y <= 40)))
                                                return 1;


                                //else
                                        //PostMessage(wndHwnd,WM_CLOSE,0,0);

        //tpr                   GetSystemTime(&systime);
        //tpr                   sprintf(cur_time,"Start Time : %d:%d:%d\nCurrent Time : %d:%d:%d\nElapsed Time :
%d:%d:%d",
        //tpr                           starttime.wHour,starttime.wMinute,starttime.wSecond,
        //tpr                           systime.wHour,systime.wMinute,systime.wSecond);

                                if ((MouseHookParam->pt.x >= 150) && (MouseHookParam->pt.x <= 186)
                                 && (MouseHookParam->pt.y >= 21) && (MouseHookParam->pt.y <= 40))
                                {
                                        _strtime(cur_time);

                                        time( &finish );
                                        elapsed = (unsigned long)difftime( finish, start );

                                        hrs = (unsigned long)elapsed / 3600;
                                        mins = (int)(elapsed - (hrs * 3600)) / 60;
                                        secs = (int)elapsed - (hrs * 3600) - (mins * 60);
                                        sprintf(timemsg, "Current  time: %s\nElapsed time:
%02d:%02d:%02d",cur_time,hrs, mins, secs);

                                        MessageBox( NULL,timemsg, "Time", MB_OK | MB_TOPMOST |
MB_TASKMODAL);

                                        wndHwnd = FindWindow(NULL, "Time");
                        }
```

```
#endif  // Delete ---------------------------------- [End]


                        }
                default:
                        wsprintf((LPSTR)szFilterLine[MOUSEINDEX],
                        "MOUSE\t\tWnd:%d Point:%d %d\t%s %s",MouseHookParam->hwnd,
                        MouseHookParam->pt.x,MouseHookParam->pt.y,
                        szMessageString(wParam),(LPSTR)szType);
    wsprintf((LPSTR)szFilterLine[MOUSEINDEX],
                "MOUSE\t\tWnd:%d Point:%d %d\t%s %s",MouseHookParam->hwnd,
                MouseHookParam->pt.x,MouseHookParam->pt.y,
                        szMessageString(wParam),(LPSTR)szType);

//      hDC = GetDC(hwndMain);
//      TabbedTextOut(hDC, 1, nLineHeight * MOUSEINDEX,
//                (LPSTR)szFilterLine[MOUSEINDEX], strlen(szFilterLine[MOUSEINDEX]),
//                0, NULL, 1 );
//      ReleaseDC(hwndMain, hDC);
                        break;
                }
                // save information to cheat file
//              fp_cheat = fopen("c:\\cheatfile.txt", "a");
//              fprintf(fp_cheat, "%s\n", szFilterLine[MOUSEINDEX]);
//              fclose(fp_cheat);
  }

  //
  // We looked at the message ... sort of processed it but since we are
  // looking we will pass all messages on to CallNextHookEx.
  //
  return( CallNextHookEx(hhookHooks[MOUSEINDEX], nCode, wParam, lParam)),
}

//---------------------------------------------------------------------------
// SysMsgFilterFunc
//
// Filter function for the WH_SYSMSGFILTER
//
//---------------------------------------------------------------------------
LRESULT CALLBACK SysMsgFilterFunc (int nCode, WPARAM wParam, LPARAM lParam )
{
  MSG *lpMsg;
//  HDC hDC;

  if ( nCode >= 0 )
  {
    if ( nCode == MSGF_DIALOGBOX )
                        strcpy(szType,"Dialog");
    else
                        strcpy(szType,"Menu");

                switch(nCode)
                {
                case MSGF_DIALOGBOX:
                        strcat(szType, "_Dialog");
                        break;
                case MSGF_MENU:
                        strcat(szType, "_Menu");
                        break;
                case MSGF_NEXTWINDOW:
                        strcat(szType, "_AltTab");
                        break;
                case MSGF_SCROLLBAR:
                        strcat(szType, "_ScrollBar");
                        break;
                }
```

```
            lpMsg = (MSG *) lParam;
    wsprintf((LPSTR)szFilterLine[SYSMSGFILTERINDEX],
                        "SYSMSGFILTER\t%s\tWnd:%d Time:%d  Point:%d %d %s                    ",
                (LPSTR)szType, lpMsg->hwnd, lpMsg->time,
                lpMsg->pt.x, lpMsg->pt.y, szMessageString(lpMsg->message));

//    hDC = GetDC(hwndMain);
//    TabbedTextOut(hDC, 1, nLineHeight * SYSMSGFILTERINDEX,
//              (LPSTR)szFilterLine[SYSMSGFILTERINDEX],
//              strlen(szFilterLine[SYSMSGFILTERINDEX]), 0, NULL, 1 );
//    ReleaseDC(hwndMain, hDC);

            // save information to cheat file
            //fp_cheat = fopen("c:\\cheatfile.txt", "a");
            //fprintf(fp_cheat, "SYSMSG %s\n", szFilterLine[SYSMSGFILTERINDEX]);
            //fclose(fp_cheat);
  }

  //
  // We looked at the message ... sort of processed it but since we are
  // looking we will pass all messages on to CallNextHookEx.
  //
  return( CallNextHookEx(hhookHooks[SYSMSGFILTERINDEX], nCode, wParam, lParam));
}

//--------------------------------------------------------------------------
// SysMsgFilterFunc
//
// Filter function for the WH_SYSMSGFILTER
//
//--------------------------------------------------------------------------
LRESULT CALLBACK DebugFilterFunc (int nCode, WPARAM wParam, LPARAM lParam )
{
  PDEBUGHOOKINFO pDebugHook;
//  HDC hDC;
  static int Called=0;

  if ( nCode >= 0 )
  {
            pDebugHook = (PDEBUGHOOKINFO) lParam;
            wsprintf((LPSTR)szFilterLine[DEBUGFILTERINDEX],
                        "DEBUGFILTER\tCalled %d Times                                    ",
                        ++Called);

//    hDC = GetDC(hwndMain);
//            TabbedTextOut(hDC, 1, nLineHeight * DEBUGFILTERINDEX,
//                        (LPSTR)szFilterLine[DEBUGFILTERINDEX],
//                        strlen(szFilterLine[DEBUGFILTERINDEX]), 0, NULL, 1 );
//    ReleaseDC(hwndMain, hDC);

            // save information to cheat file
            //fp_cheat = fopen("c:\\cheatfile.txt", "a");
            //fprintf(fp_cheat, "DBG %s\n", szFilterLine[DEBUGFILTERINDEX]);
            //fclose(fp_cheat);
  }

  //
  // We looked at the message ... sort of processed it but since we are
  // looking we will pass all messages on to CallNextHookEx.
  //
  return( CallNextHookEx(hhookHooks[SYSMSGFILTERINDEX], nCode, wParam, lParam));
}

//--------------------------------------------------------------------------
// SysMsgFilterFunc
//
// Filter function for the WH_SHELL
//
//--------------------------------------------------------------------------
```

```
LRESULT CALLBACK ShellFilterFunc (int nCode, WPARAM wParam, LPARAM lParam )
{
//  HDC hDC;
  static int Called=0,

  if ( nCode >= 0 )
  {
    switch ( nCode )
            {
            case HSHELL_ACTIVATESHELLWINDOW:
              wsprintf((LPSTR)szFilterLine[SHELLFILTERINDEX],
                              "WH_SHELL,HSHELL_ACTIVATESHELLWINDOW" );
              break;
            case HSHELL_GETMINRECT:
                    // wParam is handle of window being maximized or minimized
                    // lParam contains address of RECT that receives corrdinates
              wsprintf((LPSTR)szFilterLine[SHELLFILTERINDEX],
                              "WH_SHELL,HSHELL_GETMINRECT" );
              break;
            case HSHELL_LANGUAGE:
                    // keyboard language was changed or new keyboard layout loaded
              wsprintf((LPSTR)szFilterLine[SHELLFILTERINDEX],
                              "WH_SHELL,HSHELL_LANGUAGE" );
              break;
            case HSHELL_REDRAW:
                    // wParam contains handle of window in taskbar that has been redrawn
              wsprintf((LPSTR)szFilterLine[SHELLFILTERINDEX],
                              "WH_SHELL,HSHELL_REDRAW" );
              break;
            case HSHELL_TASKMAN:
                    // user has selected task list
                    // wParam is undefined and must be ignored
              wsprintf((LPSTR)szFilterLine[SHELLFILTERINDEX],
                              "WH_SHELL,HSHELL_TASKMAN" );
              break;
            case HSHELL_WINDOWACTIVATED:
                    // wParam contains handle of activated top-level, unowned window
              wsprintf((LPSTR)szFilterLine[SHELLFILTERINDEX],
                              "WH_SHELL,HSHELL_WINDOWACTIVATED" );
              break;
            case HSHELL_WINDOWCREATED:
                    // wParam is handle of top-level, unowned window created
              wsprintf((LPSTR)szFilterLine[SHELLFILTERINDEX],
                              "WH_SHELL,HSHELL_WINDOWCREATED" );
              break;
            case HSHELL_WINDOWDESTROYED:
                    // wParam is handle of top-level, unowned window destroyed
              wsprintf((LPSTR)szFilterLine[SHELLFILTERINDEX],
                              "WH_SHELL,HSHELL_WINDOWDESTROYED" );
              break;

            // save information to cheat file
            //fp_cheat = fopen("c:\\cheatfile.txt", "a");
            //fprintf(fp_cheat, "SHELL %s\n", szFilterLine[SHELLFILTERINDEX]);
            //fclose(fp_cheat);
            }

  }

  //
  // We looked at the message ... sort of processed it but since we are
  // looking we will pass all messages on to CallNextHookEx.
  //
  return( CallNextHookEx(hhookHooks[SYSMSGFILTERINDEX], nCode, wParam, lParam));
}


//---------------------------------------------------------------------------
// MessageString
//
```

```
// Function to load string from the STRINGTABLE
//
//-------------------------------------------------------------------------
char *szMessageString(int ID)
{
  static char szBuffer[256];

  if ( LoadString(hInstance, ID, szBuffer, 255) == 0)
                        strcpy(szBuffer,"Unknown Message");

  return (szBuffer);

}


// Who                  : Robin wei
// Date                 : 00-8-24 17:13:24
// Reason   : Replace the ole one with the one from Admin Hooks32
// Modify ------------------------------------ [Begin]
#define MsgBox(msg) ;
            BOOL CALLBACK CAPIDecryptFile(PCHAR szSource, PCHAR szDestination, PCHAR szPassword)
            {
              FILE *hSource     = NULL;
              FILE *hDestination = NULL;
              INT eof = 0;
                        int test = 0;
              HCRYPTPROV hProv   = 0;
              HCRYPTKEY hKey     = 0;
              HCRYPTHASH hHash   = 0;

              PBYTE pbKeyBlob = NULL;
              DWORD dwKeyBlobLen;

              PBYTE pbBuffer = NULL;
              DWORD dwBlockLen;
              DWORD dwBufferLen;
              DWORD dwCount;
                        LPVOID lpMsgBuf;
                        WORD PassLen;
                        char szSavedPass[256];
              BOOL status = FALSE;

                        MsgBox(szSource);
                        MsgBox(szDestination);
              // Open source file.
              if((hSource = fopen(szSource,"rb")) == NULL) {
                                  FormatMessage(       FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
                                                                                                  NULL,
                                                                                                  GetLastError(),

            MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
                                                                                                  (LPTSTR) &lpMsgBuf,    0,
NULL );// Display the string.
                                  MsgBox(lpMsgBuf);
                                  LocalFree(lpMsgBuf);
                goto done;
              }
                        test = 0;
                        MsgBox("Before create file");
              // Open destination file.
              if((hDestination = fopen(szDestination,"wb")) == NULL) {
                                  MsgBox("Error opening Plaintext file!\n");
                goto done;
              }
                        MsgBox("before Get Key create file");
                        test = 0;

              // Get handle to the default provider.
              if(!CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_FULL, 0)) {
```

```
                    // new code goes here
                    // Fail if no keyset being create before . Try to create one
                    if(!CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_FULL, CRYPT_NEWKEYSET )) {

                            char err[1024];
                            sprintf(err,"Error %x during CryptAcquireContext!\n", GetLastError());
                            MsgBox(err);

                            goto done;
                    }
}
        test = 0;

if(strcmpi(szPassword,"") == 0) {
        // Decrypt the file with the saved session key.

        // Read key blob length from source file and allocate memory.
        fread(&dwKeyBlobLen, sizeof(DWORD), 1, hSource);
        if(ferror(hSource) || feof(hSource)) {
                MsgBox("Error reading file header!\n");
          goto done;
        }
        if((pbKeyBlob = (unsigned char*)malloc(dwKeyBlobLen)) == NULL) {
          MsgBox("Out of memory or improperly formatted source file!\n");
          goto done;
        }

        // Read key blob from source file.
        fread(pbKeyBlob, 1, dwKeyBlobLen, hSource);
        if(ferror(hSource) || feof(hSource)) {
          MsgBox("Error reading file header!\n");
          goto done;
        }

        // Import key blob into CSP.
        if(!CryptImportKey(hProv, pbKeyBlob, dwKeyBlobLen, 0, 0, &hKey)) {
          MsgBox("Error %x during CryptImportKey!\n", GetLastError());
          goto done;
        }
} else {
        // Decrypt the file with a session key derived from a password.
        test = 0;
        // Create a hash object.
        if(!CryptCreateHash(hProv, CALG_MD5, 0, 0, &hHash)) {
          MsgBox("Error %x during CryptCreateHash!\n", GetLastError());
          goto done;
        }

        test = 0;

        // Hash in the password data.
        if(!CryptHashData(hHash,(unsigned char*) szPassword, strlen(szPassword), 0)) {
          MsgBox("Error %x during CryptHashData!\n", GetLastError());
          goto done;
        }

        test = 0;

        // Derive a session key from the hash object.
        if(!CryptDeriveKey(hProv, ENCRYPT_ALGORITHM, hHash, 0, &hKey)) {
          MsgBox("Error %x during CryptDeriveKey!\n", GetLastError());
          goto done;
        }

        test = 0;

        // Destroy the hash object.
        CryptDestroyHash(hHash);
        hHash = 0;
}
```

```
// Determine number of bytes to decrypt at a time. This must be a multiple
// of ENCRYPT_BLOCK_SIZE.
dwBlockLen = 1000 - 1000 % ENCRYPT_BLOCK_SIZE;
dwBufferLen = dwBlockLen;

// Allocate memory.
if((pbBuffer = (unsigned char *)malloc(dwBufferLen)) == NULL) {
        MsgBox("Out of memory!\n");
        goto done;
}
        memset(pbBuffer,'\0',1000);
        test = 0;
// 8/9/2000
// Robin.Wei
// ----------------------------------------------------------------------Begin
if(strcmpi(szPassword,"")) // User provide Password , we encrypte it first ,
                                                                        // so that we can determind whether the
Decrypted is right

{
        // Read up to 'dwBlockLen' bytes from source file.
        dwCount = fread(pbBuffer, 1, dwBlockLen, hSource);
        if(ferror(hSource)) {
          MsgBox("Error reading Ciphertext!\n");
      goto done;
 }

// Decrypt data
if(!CryptDecrypt(hKey, 0, FALSE, 0, pbBuffer, &dwCount)) {
   MsgBox("Error %x during CryptDecrypt!\n", GetLastError());
   goto done;
}


        PassLen = MAKEWORD(*pbBuffer,*(pbBuffer+1));
        if(PassLen >= dwCount-2 || PassLen <=0)
                goto done;
        ZeroMemory(szSavedPass,sizeof(szSavedPass));

        memcpy(szSavedPass,pbBuffer+2,PassLen);
        if(strcmpi(szPassword,szSavedPass)==0)
        {

                        // Write data to destination file.
                fwrite(pbBuffer+2+PassLen, 1, dwCount-2-PassLen, hDestination);
                if(ferror(hDestination)) {
                        MsgBox("Error writing Plaintext!\n");
                                goto done;
                }
        }else
        {
                goto done;
        }

}
// ----------------------------------------------------------------------End

// Decrypt source file and write to destination file.
do {
        // Read up to 'dwBlockLen' bytes from source file.
        dwCount = fread(pbBuffer, 1, dwBlockLen, hSource);
        if(ferror(hSource)) {
          MsgBox("Error reading Ciphertext!\n");
    goto done;
  }
        test = 0;
        eof = feof(hSource);

  // Decrypt data
```

```
            if(!CryptDecrypt(hKey, 0, eof, 0, pbBuffer, &dwCount)) {
              MsgBox("Error %x during CryptDecrypt!\n", GetLastError());
              goto done;
            }

            // Write data to destination file.
                fwrite(pbBuffer, 1, dwCount, hDestination);
                if(ferror(hDestination)) {
                  MsgBox("Error writing Plaintext!\n");
              goto done;
            }
                test = 0;
          } while(!feof(hSource));

          status = TRUE;

          printf("OK\n");

          done:

          // Close files.
          if(hSource) fclose(hSource);
          if(hDestination) fclose(hDestination);

          // Free memory.
          if(pbKeyBlob) free(pbKeyBlob);
          if(pbBuffer) free(pbBuffer);

          // Destroy session key.
          if(hKey) CryptDestroyKey(hKey);

          // Destroy hash object.
  if(hHash) CryptDestroyHash(hHash);

  // Release provider handle.
  if(hProv) CryptReleaseContext(hProv, 0);

  return(status);
}
// Modify ------------------------------------ [End]



// Who              : Robin wei
// Date             : 00-8-24 17:12:49
// Reason  : This funtion seem not correct. Replace it with  the one from Admin\Hook32
#if 0 // Delete ---------------------------------- [Begin]
          BOOL CALLBACK CAPIDecryptFile(PCHAR szSource, PCHAR szDestination, PCHAR szPassword)
          {
                BOOL eof = FALSE;
                DWORD dwErrCode;
                HANDLE hSourceFile;
                HANDLE hTargetFile;

            HCRYPTPROV hProv   = 0;
            HCRYPTKEY hKey     = 0;
            HCRYPTKEY hXchgKey = 0;
            HCRYPTHASH hHash   = 0;

            PBYTE pbKeyBlob = NULL;
            DWORD dwKeyBlobLen;

            PBYTE pbBuffer = NULL;
            DWORD dwBlockLen;
            DWORD dwBufferLen;
            DWORD dwCount;
                DWORD dwBytesWritten;
                DWORD dwBytesRead;
```

```
        BOOL bWriteRetVal = FALSE;
    BOOL status = FALSE;

        LPVOID lpMsgBuf;
        int test = 0;

    // Open source file.

        hSourceFile = CreateFile(szSource,
                                                                GENERIC_READ,
                                                                0,
                                                                NULL,
                                                                OPEN_EXISTING,
                                                                FILE_ATTRIBUTE_NORMAL,
                                                                NULL);

        dwErrCode = GetLastError();

        if(hSourceFile== INVALID_HANDLE_VALUE)
        {
                // new code goes here
                FormatMessage(      FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
                                                                NULL,
                                                                dwErrCode,

    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
                                                                (LPTSTR)
&lpMsgBuf,   0,   NULL );// Display the string.
                LocalFree(lpMsgBuf);

                return FALSE;

        }
        test = 0;
        hTargetFile = CreateFile(szDestination,
                                                                GENERIC_WRITE,
                                                                0,
                                                                NULL,
                                                                CREATE_ALWAYS,
                                                                FILE_ATTRIBUTE_NORMAL,
                                                                NULL);

        dwErrCode = GetLastError();

        if(hTargetFile == INVALID_HANDLE_VALUE)
        {
                // new code goes here
                FormatMessage(      FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
                                                                NULL,
                                                                dwErrCode,

    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
                                                                (LPTSTR)
&lpMsgBuf,   0,   NULL );// Display the string.
                LocalFree(lpMsgBuf);
                return FALSE;
        }
        test = 0;

    // Get handle to the default provider.
    if(!CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_FULL, 0)) {
      printf("Error %x during CryptAcquireContext!\n", GetLastError());
      return FALSE;
    }
        test = 0;

    if(strcmpi(szPassword,"") == 0) {

        // Decrypt the file with the saved session key.
        // Read key blob length from source file and allocate memory.
```

```
BOOL bReadRetVal = ReadFile(hSourceFile,
                                                        &dwKeyBlobLen,
                                                        sizeof(DWORD),
                                                        &dwBytesRead,
                                                        NULL);

dwErrCode = GetLastError();
if(!(bReadRetVal > 0))
{
        FormatMessage(      FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
                                                        NULL,
                                                        dwErrCode,

    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
                                                        (LPTSTR)
&lpMsgBuf,   0,   NULL );// Display the string.
                LocalFree(lpMsgBuf);
                return FALSE;
}
test = 0;
if((pbKeyBlob = (unsigned char*)malloc(dwKeyBlobLen)) == NULL) {
   printf("Out of memory or improperly formatted source file!\n");
   return FALSE;
}
memset(pbKeyBlob,NULL,dwKeyBlobLen);
test = 0;
// Read key blob from source file.
bReadRetVal = ReadFile(hSourceFile,
                                                        pbKeyBlob,
                                                        dwKeyBlobLen,
                                                        &dwBytesRead,
                                                        NULL);

dwErrCode = GetLastError();
if(!(bReadRetVal > 0))
{
        FormatMessage(      FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
                                                        NULL,
                                                        dwErrCode,

    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
                                                        (LPTSTR)
&lpMsgBuf,   0,   NULL );// Display the string.
                LocalFree(lpMsgBuf);
                return FALSE;
}
test = 0;
// Import key blob into CSP.
if(!CryptImportKey(hProv, pbKeyBlob, dwKeyBlobLen, 0, 0, &hKey)) {
   printf("Error %x during CryptImportKey!\n", GetLastError());
   return FALSE;
}
} else {
// Decrypt the file with a session key derived from a password.

// Create a hash object.
if(!CryptCreateHash(hProv, CALG_MD5, 0, 0, &hHash)) {
   printf("Error %x during CryptCreateHash!\n", GetLastError());
   return FALSE;
}
test = 0;
// Hash in the password data.
if(!CryptHashData(hHash,(unsigned char*) szPassword, strlen(szPassword), 0)) {
   printf("Error %x during CryptHashData!\n", GetLastError());
   return FALSE;
}
test = 0;
// Derive a session key from the hash object.
if(!CryptDeriveKey(hProv, ENCRYPT_ALGORITHM, hHash, 0, &hKey)) {
```

```
                printf("Error %x during CryptDeriveKey!\n", GetLastError());
                return FALSE;
            }
            test = 0;
            // Destroy the hash object.
            CryptDestroyHash(hHash);
            hHash = 0;
        }

        // Determine number of bytes to decrypt at a time. This must be a multiple
        // of ENCRYPT_BLOCK_SIZE.
        dwBlockLen = 1000 - 1000 % ENCRYPT_BLOCK_SIZE;
        dwBufferLen = dwBlockLen;

        // Allocate memory.
        if((pbBuffer = (unsigned char *)malloc(dwBufferLen)) == NULL) {
            printf("Out of memory!\n");
            return FALSE;
        }
            memset(pbBuffer,NULL,dwBufferLen);
            test = 0;
        // Decrypt source file and write to destination file.
        do {
            // Read up to 'dwBlockLen' bytes from source file.

                BOOL bReadRetVal = ReadFile(hSourceFile,
                                                                    pbBuffer,
                                                                    dwBlockLen,
                                                                    &dwBytesRead,
                                                                    NULL);

            dwErrCode = GetLastError();
            if(!(bReadRetVal > 0))
            {
                FormatMessage(  FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
                                                                    NULL,
                                                                    dwErrCode,

    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
                                                                    (LPTSTR)
&lpMsgBuf,  0,  NULL );// Display the string.
                LocalFree(lpMsgBuf);
                return FALSE;
            }
            test = 0;

            if(dwBytesRead == 0)
                eof = TRUE;
            else
                eof = FALSE;

          // Decrypt data
          if(!CryptDecrypt(hKey, 0, eof, 0, pbBuffer, &dwBytesRead)) {
            printf("Error %x during CryptDecrypt!\n", GetLastError());
            return FALSE;
          }
                test = 0;
          // Write data to destination file.

            bWriteRetVal = WriteFile(   hTargetFile,
                                                                    pbBuffer,
                                                                    dwBytesRead,
                                                                    &dwBytesWritten,
                                                                    NULL);

            dwErrCode = GetLastError();
            if(!(bWriteRetVal > 0))
            {
                FormatMessage(  FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
                                                                    NULL,
```

```
                                                                    dwErrCode,

        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT). // Default language
                                                                    (LPTSTR)
&lpMsgBuf,   0,   NULL );// Display the string.
                                LocalFree(lpMsgBuf);
                                return FALSE;
                        }

                        } while(!(dwBytesRead > 0));
                        test = 0;
              status = TRUE;

              // Close files.
              if(hSourceFile) CloseHandle(hSourceFile);
              if(hTargetFile) CloseHandle(hTargetFile);

              // Free memory.
              if(pbKeyBlob) free(pbKeyBlob);
              if(pbBuffer) free(pbBuffer);

              // Destroy session key.
  if(hKey) CryptDestroyKey(hKey);

  // Destroy hash object.
  if(hHash) CryptDestroyHash(hHash);

  // Release provider handle.
  if(hProv) CryptReleaseContext(hProv, 0);

  return(status);
}
#endif  // Delete ---------------------------------- [End]


BOOL CALLBACK CAPIEncryptFile(PCHAR szSource,  PCHAR szDestination, PCHAR szPassword)
{
  BOOL eof = FALSE;
          DWORD dwErrCode;
          HANDLE hSourceFile;
          HANDLE hTargetFile;

  HCRYPTPROV hProv   = 0;
  HCRYPTKEY hKey     = 0;
  HCRYPTKEY hXchgKey = 0;
  HCRYPTHASH hHash   = 0;
          BOOL bRetVal;
  PBYTE pbKeyBlob = NULL;
  DWORD dwKeyBlobLen;

  PBYTE pbBuffer = NULL;
  DWORD dwBlockLen;
  DWORD dwBufferLen;
          DWORD dwBytesWritten;
          DWORD dwBytesRead;
          WORD PassLen ;
          BOOL bWriteRetVal = FALSE;
  BOOL status = FALSE;

          LPVOID lpMsgBuf;
          int test =0 ;
  // Open source file.

          hSourceFile = CreateFile(szSource,
                                                                    GENERIC_READ,
                                                                    0,
                                                                    NULL,
                                                                    OPEN_EXISTING,
                                                                    FILE_ATTRIBUTE_NORMAL,
                                                                    NULL);
```

```
        dwErrCode = GetLastError();
        if(hSourceFile== INVALID_HANDLE_VALUE)
        {
                // new code goes here
                FormatMessage(      FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
                                                                        NULL,
                                                                        dwErrCode,

        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
                                                                        (LPTSTR) &lpMsgBuf,   0,
NULL );// Display the string.
                LocalFree(lpMsgBuf);
                return FALSE;

        }
        test = 0;
        hTargetFile = CreateFile(szDestination,
                                                                GENERIC_WRITE,
                                                                0,
                                                                NULL,
                                                                CREATE_ALWAYS,
                                                                FILE_ATTRIBUTE_NORMAL,
                                                                NULL);

        dwErrCode = GetLastError();

        if(hTargetFile == INVALID_HANDLE_VALUE)
        {
                // new code goes here
                        FormatMessage(      FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
                                                                NULL,
                                                                dwErrCode,

        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
                                                                (LPTSTR) &lpMsgBuf,   0,
NULL );// Display the string.
                                MsgBox(lpMsgBuf);
                                LocalFree(lpMsgBuf);
                        return FALSE;
        }
        test = 0;
  // Get handle to the default provider.
  if(!CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_FULL, 0)) {
                // new code goes here
                        FormatMessage(      FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
                                                                NULL,
                                                                GetLastError(),

        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
                                                                (LPTSTR) &lpMsgBuf,   0,
NULL );// Display the string.
                                MsgBox(lpMsgBuf);
                                LocalFree(lpMsgBuf);
                        return FALSE;
    return FALSE;
  }
        test = 0;
  if(strcmpi(szPassword,"") == 0) {

        // Encrypt the file with a random session key.
        // Create a random session key.
        if(!CryptGenKey(hProv, ENCRYPT_ALGORITHM, CRYPT_EXPORTABLE, &hKey)) {
          MsgBox("Error %x during CryptGenKey!\n", GetLastError());
          return FALSE;
        }
        test = 0;
        // Get handle to key exchange public key.
        if(!CryptGetUserKey(hProv, AT_KEYEXCHANGE, &hXchgKey)) {
```

```
            MsgBox("Error %x during CryptGetUserKey!\n", GetLastError());
            return FALSE;
        }
        test = 0;
                // Determine size of the key blob and allocate memory.
                if(!CryptExportKey(hKey, hXchgKey, SIMPLEBLOB, 0, NULL, &dwKeyBlobLen))
                {
                        MsgBox("Error %x computing blob length!\n", GetLastError());
                        return FALSE;
                }
                test = 0;
                if((pbKeyBlob = (unsigned char*)malloc(dwKeyBlobLen)) == NULL)
                {
                        MsgBox("Out of memory!\n");
                        return FALSE;
                }
                test = 0,
                // Export session key into a simple key blob.
                if(!CryptExportKey(hKey, hXchgKey, SIMPLEBLOB, 0, pbKeyBlob, &dwKeyBlobLen))
                {
                        MsgBox("Error %x during CryptExportKey!\n", GetLastError());
                        return FALSE;
                }
                test = 0;
        // Release key exchange key handle.
        CryptDestroyKey(hXchgKey);
        hXchgKey = 0;

        // Write size of key blob to destination file.
        bRetVal = WriteFile(hTargetFile,&dwKeyBlobLen,sizeof(DWORD),&dwBytesWritten,NULL);
        if(!(bRetVal > 0))
        {
                MsgBox("Error write file");
                return FALSE;
        }
        test = 0;
        // Write key blob to destination file.
         bWriteRetVal = WriteFile(hTargetFile,pbKeyBlob,dwKeyBlobLen,&dwBytesWritten,NULL);
        if(!(bWriteRetVal > 0))
        {
                MsgBox("Error Write File");
                return FALSE;
        }

} else {
        // Encrypt the file with a session key derived from a password.

        // Create a hash object.
        if(!CryptCreateHash(hProv, CALG_MD5, 0, 0, &hHash)) {
           MsgBox("Error %x during CryptCreateHash!\n", GetLastError());
           return FALSE;
        }
        test = 0;
        // Hash in the password data.
        if(!CryptHashData(hHash, (unsigned char*)szPassword, strlen(szPassword), 0)) {
           MsgBox("Error %x during CryptHashData!\n", GetLastError());
           return FALSE;
        }
        test = 0;
        // Derive a session key from the hash object.
        if(!CryptDeriveKey(hProv, ENCRYPT_ALGORITHM, hHash, 0, &hKey)) {
           MsgBox("Error %x during CryptDeriveKey!\n", GetLastError());
           return FALSE;
        }
        test = 0;
        // Destroy the hash object.
        CryptDestroyHash(hHash);
        hHash = 0;
}
```

```
// Determine number of bytes to encrypt at a time. This must be a multiple
// of ENCRYPT_BLOCK_SIZE.
dwBlockLen = 1000 - 1000 % ENCRYPT_BLOCK_SIZE;

// Determine the block size. If a block cipher is used this must have
// room for an extra block.
if(ENCRYPT_BLOCK_SIZE > 1) {
        dwBufferLen = dwBlockLen + ENCRYPT_BLOCK_SIZE;
} else {
        dwBufferLen = dwBlockLen;
}

// Allocate memory.
if((pbBuffer = (unsigned char*)malloc(dwBufferLen)) == NULL) {
                MsgBox("Out of memory!\n");
                return FALSE;
}
        memset(pbBuffer,0,dwBufferLen);
        test = 0;
        // 8/9/2000
        // Robin.Wei
        // ----------------------------------------------------------------------Begin
        if(strcmpi(szPassword,"")) // User provide Password , we encrypte it first ,
                                                                        // so that we can determind whether the
Decrypted is right

        {

                PassLen = strlen(szPassword);
                pbBuffer[0]=LOBYTE( PassLen);
                pbBuffer[1]=HIBYTE(PassLen);
                strcpy(pbBuffer+2,szPassword);
                dwBytesRead = 2+PassLen;
                        // Encrypt data
                if(!CryptEncrypt(hKey, 0, FALSE, 0, pbBuffer, &dwBytesRead, dwBufferLen))
                {
                        dwErrCode = GetLastError();
                        FormatMessage(      FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
                                                                                NULL,
                                                                                dwErrCode,

        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default !anguage
                                                                                (LPTSTR)
&lpMsgBuf,   0,   NULL );// Display the string.
                        MsgBox(lpMsgBuf);
                        LocalFree(lpMsgBuf);
                        return FALSE;
                        }
                test = 0;
                // Write data to destination file.
                bWriteRetVal = WriteFile(       hTargetFile,
                                                                        pbBuffer,
                                                                        dwBytesRead,
                                                                        &dwBytesWritten,
                                                                        NULL);

                dwErrCode = GetLastError();
                if(!(bWriteRetVal > 0))
                {
                        FormatMessage(      FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
                                                                                NULL,
                                                                                dwErrCode,

        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
                                                                                (LPTSTR)
&lpMsgBuf,   0,   NULL );// Display the string.
                        MsgBox(lpMsgBuf);
                        LocalFree(lpMsgBuf);
                        return FALSE;
```

```
                }

        }
        // ----------------------------------------------------------------End
// Encrypt source file and write to Source file.
do {
                // Read up to 'dwBlockLen' bytes from source file.

                BOOL bReadRetVal = ReadFile(hSourceFile,
                                                                        pbBuffer,
                                                                        dwBlockLen,
                                                                        &dwBytesRead,
                                                                        NULL);

                dwErrCode = GetLastError();
                if(!(bReadRetVal > 0))
                {
                        FormatMessage(      FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
                                                                                NULL,
                                                                                dwErrCode,

        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
                                                                                (LPTSTR)
&lpMsgBuf,   0,   NULL );// Display the string.
                        MsgBox(lpMsgBuf);
                        LocalFree(lpMsgBuf);
                        return FALSE;
                }
                test = 0;
                if(dwBytesRead == 0)
                {
                        eof = TRUE;
                        break;
                }
                else
                        eof = FALSE;

                        // Encrypt data
                if(!CryptEncrypt(hKey, 0, eof, 0, pbBuffer, &dwBytesRead, dwBufferLen))
                {
                        dwErrCode = GetLastError();
                        FormatMessage(      FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
                                                                                NULL,
                                                                                dwErrCode,

        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
                                                                                (LPTSTR)
&lpMsgBuf,   0,   NULL );// Display the string.
                        MsgBox(lpMsgBuf);
                        LocalFree(lpMsgBuf);
                        return FALSE;
                        }
                test = 0;
                // Write data to destination file.
                bWriteRetVal = WriteFile(       hTargetFile,
                                                                        pbBuffer,
                                                                        dwBytesRead,
                                                                        &dwBytesWritten,
                                                                        NULL);

                dwErrCode = GetLastError();
                if(!(bWriteRetVal > 0))
                {
                        FormatMessage(      FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
                                                                                NULL,
                                                                                dwErrCode,

        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
```

```
                                                                    (LPTSTR)
&lpMsgBuf,    0,    NULL );// Display the string.
                                MsgBox(lpMsgBuf);
                                LocalFree(lpMsgBuf);
                                return FALSE;
                        }
                } while(!(dwBytesRead == 0));
                test = 0;
        status = TRUE;

        // Close files.
        if(hSourceFile) CloseHandle(hSourceFile);
        if(hTargetFile) CloseHandle(hTargetFile);

        // Free memory.
        if(pbKeyBlob) free(pbKeyBlob);
        if(pbBuffer) free(pbBuffer);

        // Destroy session key.
        if(hKey) CryptDestroyKey(hKey);

        // Release key exchange key handle.
        if(hXchgKey) CryptDestroyKey(hXchgKey);

        // Destroy hash object.
        if(hHash) CryptDestroyHash(hHash);

        // Release provider handle.
        if(hProv) CryptReleaseContext(hProv, 0);

        return(status);
}


BOOL  CALLBACK InitUser()
{
        HCRYPTPROV hProv;
        HCRYPTKEY hKey;
        CHAR szUserName[100];
        DWORD dwUserNameLen = 100;

        // Attempt to acquire a handle to the default key container.
        if(!CryptAcquireContext(&hProv, NULL, MS_DEF_PROV, PROV_RSA_FULL, 0)) {
                // Some sort of error occured.
                        CryptAcquireContext(&hProv, NULL, MS_DEF_PROV, PROV_RSA_FULL, CRYPT_DELETEKEYSET);
                        // Create default key container.
                        if(!CryptAcquireContext(&hProv, NULL, MS_DEF_PROV, PROV_RSA_FULL, CRYPT_NEWKEYSET)) {
                                LPVOID lpMsgBuf;
                                DWORD err = GetLastError();

                                FormatMessage(
                                        FORMAT_MESSAGE_ALLOCATE_BUFFER |
                                        FORMAT_MESSAGE_FROM_SYSTEM |
                                        FORMAT_MESSAGE_IGNORE_INSERTS,
                                        NULL,
                                        err,
                                        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
                                        (LPTSTR) &lpMsgBuf,
                                        0,
                                        NULL
                                );
                                // Process any inserts in lpMsgBuf.
                                // ...
                                // Display the string.
                                MessageBox( NULL, (LPCTSTR)lpMsgBuf, "Error", MB_OK | MB_ICONINFORMATION );
                                sprintf(lpMsgBuf,"Error Num: %x",err);
                                MessageBox(NULL,(LPCTSTR)lpMsgBuf,"Error",MB_OK | MB_ICONINFORMATION);

                                // Free the buffer.
                                LocalFree( lpMsgBuf );
```

```
                exit(1);
        }

        // Get name of default key container.
        if(!CryptGetProvParam(hProv, PP_CONTAINER, szUserName, &dwUserNameLen, 0)) {
                // Error getting key container name.
                szUserName[0] = 0;
        }

        printf("Create key container '%s'\n",szUserName);
}

// Attempt to get handle to signature key.
if(!CryptGetUserKey(hProv, AT_SIGNATURE, &hKey))
        {
                if(GetLastError() == NTE_NO_KEY)
                {
                        // Create signature key pair.
                        printf("Create signature key pair\n");

                        if(!CryptGenKey(hProv,AT_SIGNATURE,0,&hKey))
                        {
                                printf("Error %x during CryptGenKey!\n", GetLastError());
                                exit(1);
                        }
                        else {
                                CryptDestroyKey(hKey);
                        }
                } else {
                        printf("Error %x during CryptGetUserKey!\n", GetLastError());
                        exit(1);
                }
 }

 // Attempt to get handle to exchange key.
 if(!CryptGetUserKey(hProv,AT_KEYEXCHANGE,&hKey))
        {
                if(GetLastError()==NTE_NO_KEY)
                {
                        // Create key exchange key pair.
                        printf("Create key exchange key pair\n");

                        if(!CryptGenKey(hProv,AT_KEYEXCHANGE,0,&hKey))
                        {
                                printf("Error %x during CryptGenKey!\n", GetLastError());
                                exit(1);
                        } else
                        {
                                CryptDestroyKey(hKey);
                        }
                } else {
                        printf("Error %x during CryptGetUserKey!\n", GetLastError());
                        exit(1);
                }
 }

 CryptReleaseContext(hProv,0);

 printf("OK\n");
        return 1;
 //exit(0);

}
#include "Tlhelp32.h"
int CALLBACK CheckStart(void)
{
        int ret = -1;
        OSVERSIONINFO osver;
        HINSTANCE     hInstLib;
```

```
HANDLE      hSnapShot;
PROCESSENTRY32 procentry;
BOOL        bFlag;
BOOL bRetVal = FALSE;
BOOL bIniVal = FALSE;
UINT retval = 0;
BOOL bProcFound= FALSE;
int count;
LPTSTR ValidProcs = NULL;
HANDLE hNonValidProcs;

// ToolHelp Function Pointers.
HANDLE (WINAPI *lpfCreateToolhelp32Snapshot)(DWORD,DWORD);
BOOL (WINAPI *lpfProcess32First)(HANDLE,LPPROCESSENTRY32);
BOOL (WINAPI *lpfProcess32Next)(HANDLE,LPPROCESSENTRY32);



// Check to see if were running under Windows95 or Windows NT.
osver.dwOSVersionInfoSize = sizeof( osver );
if( !GetVersionEx( &osver ) )
{
        return ret;
}


// If Windows 95:
if( osver.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS )
{
        hInstLib = LoadLibraryA( "Kernel32.DLL" );
        if( hInstLib == NULL )
                return ret;

        // Get procedure addresses.
        // We are linking to these functions of Kernel32 explicitly, because
        // otherwise a module using this code would fail to load under Windows NT,
        // which does not have the Toolhelp32 functions in the Kernel 32.
        lpfCreateToolhelp32Snapshot= (HANDLE(WINAPI *)(DWORD,DWORD))
                GetProcAddress( hInstLib, "CreateToolhelp32Snapshot" );
        lpfProcess32First= (BOOL(WINAPI *)(HANDLE,LPPROCESSENTRY32))
                GetProcAddress( hInstLib, "Process32First" );
        lpfProcess32Next= (BOOL(WINAPI *)(HANDLE,LPPROCESSENTRY32))
                GetProcAddress( hInstLib, "Process32Next" );

        if( lpfProcess32Next == NULL || lpfProcess32First == NULL ||
                lpfCreateToolhelp32Snapshot == NULL )
        {
                FreeLibrary( hInstLib );
                return ret;
        }

        // Get a handle to a Toolhelp snapshot of the systems processes.
        hSnapShot = lpfCreateToolhelp32Snapshot( TH32CS_SNAPPROCESS, 0 );

        if( hSnapShot == INVALID_HANDLE_VALUE )
        {
                FreeLibrary( hInstLib );
return ret;
        }

        // Get the first process' information.
        procentry.dwSize = sizeof(PROCESSENTRY32);
        bFlag = lpfProcess32First( hSnapShot, &procentry );
        // While there are processes, keep looping.
        count=0;
        while( bFlag)
        {
                count ++;
                procentry.dwSize = sizeof(PROCESSENTRY32);
```

```
                        bFlag = lpfProcess32Next( hSnapShot, &procentry );
                }
                ret = count;
        }

        // Free the library.
        FreeLibrary( hInstLib );

        return ret ;
}

int CALLBACK MyEnableDbClick(int value)
{
        bEnableDBClick = value;
        return bEnableDBClick>0 ? 1:0;
}


int CALLBACK BeginThread ()
{
        DWORD ThreadID;
        OSVERSIONINFO  osver;
        osver.dwOSVersionInfoSize = sizeof( osver );
        if( !GetVersionEx( &osver ) )
        {
                return 0;
        }

        if( osver.dwPlatformId < VER_PLATFORM_WIN32_NT )
        {
                return 0;
        }
        bStart=TRUE;
        hThread=CreateThread(NULL,0,ThreadProc,0.0,&ThreadID);
        return 0;
}

DWORD WINAPI ThreadProc(LPVOID lpParameter)
{

        while(bStart)
        {

                EnumWindows(EnumWindowsProc,0);
                Sleep(500);

        }
        return 0;
}


BOOL CALLBACK EnumWindowsProc(HWND hwnd,LPARAM lParam)
{
        RemoveWindow(hwnd);
        return TRUE;
}

int  CALLBACK EndThread ()
 {
        if(hThread != NULL)
        {
                bStart=FALSE;
                WaitForSingleObject(hThread,5000);
        }

        return 0;
 }

BOOL RemoveWindow(HWND hWnd)
{
```

```
        HMODULE      hMod;
        DWORD ret;
        DWORD ProcessID,ThreadID;
        HANDLE hProcess = NULL;
        char         szFileName[ 255 ];
        HINSTANCE    hInstLib;


        BOOL (WINAPI *lpfEnumProcessModules)( HANDLE, HMODULE *, DWORD, LPDWORD );
        DWORD (WINAPI *lpfGetModuleFileNameEx)( HANDLE, HMODULE, LPTSTR, DWORD );

#ifdef _DEBUG
   fp_cheat = fopen("c:\\cheatfile.txt", "a");
#endif

        // Load library and get the procedures explicitly. We do
        // this so that we don't have to worry about modules using
        // this code failing to load under Windows 95, because
        // it can't resolve references to the PSAPI.DLL.
        hInstLib = LoadLibraryA( "PSAPI.DLL" );
        if( hInstLib == NULL )
                return FALSE;
#ifdef _DEBUG
   fprintf(fp_cheat, "CBT %s\n", "Load PSAPI.DLL OK");
#endif

        // Get procedure addresses.
        lpfEnumProcessModules = (BOOL(WINAPI *)(HANDLE, HMODULE *, DWORD, LPDWORD))
                GetProcAddress( hInstLib, "EnumProcessModules" );
        lpfGetModuleFileNameEx =(DWORD (WINAPI *)(HANDLE, HMODULE, LPTSTR, DWORD ))
                GetProcAddress( hInstLib, "GetModuleFileNameExA" );



        ZeroMemory(szFileName,sizeof(szFileName));
        ThreadID = GetWindowThreadProcessId(hWnd,&ProcessID);
        hProcess = OpenProcess( PROCESS_QUERY_INFORMATION | PROCESS_VM_READ,
                                FALSE, ProcessID );
#ifdef _DEBUG
   fprintf(fp_cheat, "CBT %s\n", "OPEN PROCESS");
#endif
        if( hProcess != NULL )
        {
#ifdef _DEBUG
                fprintf(fp_cheat, "CBT %s\n", "OPEN PROCESS OK");
#endif
                // Here we call EnumProcessModules to get only the
                // first module in the process this is important,
                // because this will be the .EXE module for which we
                // will retrieve the full path name in a second.
                if( lpfEnumProcessModules( hProcess, &hMod, sizeof( hMod ), &ret) )
                {
#ifdef _DEBUG
                        fprintf(fp_cheat, "CBT %s\n", "GET MODULE OK");
#endif
                        // Get Full pathname:
                        if( !lpfGetModuleFileNameEx( hProcess, hMod, szFileName, sizeof( szFileName ) ) )
                        {
#ifdef _DEBUG
                                fprintf(fp_cheat, "CBT %s\n", "GET MODULE NAME FAILER");
#endif
                                szFileName[0] = -1;
                        }
                }
                CloseHandle( hProcess );
        }
        FreeLibrary( hInstLib );
        if(*szFileName != -1)
        {
```

```
#ifdef _DEBUG

                fprintf(fp_cheat, "CBT %s\n", "GET MODULE NAME OK");
                fprintf(fp_cheat, "CBT %s\n", szFileName);
#endif
                strupr(szFileName);
                if(strstr(szFileName,"WINWORD.EXE") ||
                        strstr(szFileName,"SSI_STUDENT.EXE") ||
                        strstr(szFileName,"SSI_TEMP.DAT") ||
                        strstr(szFileName,"EXPLORER.EXE") ||
                        strstr(szFileName,"IEXPLORE.EXE") ||
                        strstr(szFileName,"KERNEL32.DLL") ||
//                      strstr(szFileName,"MSGSRV32.EXE") ||
//                      strstr(szFileName,"MPREXE.EXE") ||
//                      strstr(szFileName,"MSTASK.EXE") ||
//                      strstr(szFileName,"RUNONCE.EXE") ||
                        strstr(szFileName,"RPCSS.EXE") ||
//                      strstr(szFileName,"SPOOLSV.EXE") ||
                        strstr(szFileName,"SSI_TIMER.DLL") ||
                        strstr(szFileName,"WINLOGON.EXE"))
//                      strstr(szFileName,"CSRSS.EXE") ||
//                      strstr(szFileName,"WINMGMT.EXE") ||
//                      strstr(szFileName,"MSDEV.EXE") ||
//                      strstr(szFileName,"HOOKS32.EXE"))
                {
#ifdef _DEBUG

                        fprintf(fp_cheat, "CBT %s\n", "RETURN TRUE");
                        fclose(fp_cheat);
#endif
                        if(strstr(szFileName,"EXPLORER.EXE") )
                        {
                                char tmpStr[80];
                                GetClassName(hWnd,tmpStr,80);

                                if(stricmp(tmpStr,"CabinetWClass")==0 ||
                                        stricmp(tmpStr,"IEFrame")==0||
                                        stricmp(tmpStr,"#32770")==0 ||
                                        stricmp(tmpStr,"ExploreWClass")==0 )
                                {
        #ifdef _DEBUG
                                        fprintf(fp_cheat, "CBT EXPLORER 222: %s\n", tmpStr);
                                        fclose(fp_cheat);
        #endif
                                        if(GetWindowLong(hWnd,GWL_STYLE) & WS_VISIBLE)
                                        {
                                                PostMessage(hWnd,WM_CLOSE,0,0);
                                        }

                                        return TRUE;
                                }

                        }
                        return TRUE;
                }
                else
                {
                        char tmpStr[80];
                        GetClassName(hWnd,tmpStr,80);
                        if(stricmp(tmpStr,"Shell_TrayWnd")==0||
                                stricmp(tmpStr,"progman")==0)
                        {
                                return TRUE;
                        }
#ifdef _DEBUG
                        fprintf(fp_cheat, "CBT kkkkkkkkkkkkkkk : %s\n", tmpStr);
                        fprintf(fp_cheat, "CBT %s\n", "CHECK VISIBLE");
#endif
                        if(GetWindowLong(hWnd,GWL_STYLE) & WS_VISIBLE)
                        {
```

```
                                        PostMessage(hWnd,WM_CLOSE,0,0);
#ifdef _DEBUG
                                        fprintf(fp_cheat, "CBT %s\n", "Killed");

                                        fclose(fp_cheat);
#endif
                                        return TRUE;
                                }
                        }
                }
                return TRUE;
}
int CALLBACK CheckUserSid(LPSTR outDomainName,LPSTR outUserName)
{
                PSID oldSid;
                HANDLE TokenHandle;
                DWORD ReturnLength;
                TOKEN_USER *tokenUser;
                LONG lRetErrorCode;
                HKEY NewKey = NULL;
                BYTE * lpBuf;
                LONG length;
                LPVOID lpMsgBuf;
                OSVERSIONINFO  osver;
                char UserName[30];
                char DomainName[30];
                DWORD size;
                DWORD psize;

                *outDomainName = 0;
                *outUserName=0;


                osver.dwOSVersionInfoSize = sizeof( osver );
                if( !GetVersionEx( &osver ) )
                {
                        return 0;
                }
                if( osver.dwPlatformId < VER_PLATFORM_WIN32_NT )
                {

                        TCHAR ttt[255];
                        LONG cbTTT=255;
                        GetUserName(ttt,&cbTTT);
                        size=30;
                        psize=30;
                        lRetErrorCode=RegOpenKey(    HKEY_LOCAL_MACHINE,

                "Software\\Microsoft\\Windows\\CurrentVersion\\Winlogon",
                                                                        &NewKey);
                        RegQueryValueEx(NewKey,"SavedUserName",0,NULL,(unsigned char *)UserName,&size);
                        strcpy(outDomainName,"");
                        strcpy(outUserName,UserName);
                        RegCloseKey(NewKey);
                        if(strcmp(UserName,ttt)==0)
                                return 1;
                        else
                                return 0;
                }

                if(!OpenProcessToken( OpenProcess(PROCESS_ALL_ACCESS,FALSE,GetCurrentProcessId()), // handle to process
                                                TOKEN_ALL_ACCESS,  // desired access to process
                                                &TokenHandle   // handle to open access token
                                                ))

                {
                        LPVOID lpMsgBuf;
                        FormatMessage(
                                FORMAT_MESSAGE_ALLOCATE_BUFFER |
                                FORMAT_MESSAGE_FROM_SYSTEM |
                                FORMAT_MESSAGE_IGNORE_INSERTS,
```

```
                    NULL,
                    GetLastError(),
                    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
                    (LPTSTR) &lpMsgBuf,
                    0,
                    NULL
            );

            MessageBox( NULL, (LPCTSTR)lpMsgBuf, "Error", MB_OK | MB_ICONINFORMATION );
            // Free the buffer.
            LocalFree( lpMsgBuf );

        }
        GetTokenInformation(  TokenHandle,                  // handle to access token
                              TokenUser, // token type
                              NULL,                 // buffer
                              0,              // size of buffer
                              &ReturnLength                 // required buffer size
                             );

        tokenUser = malloc(ReturnLength);
        if(!GetTokenInformation(  TokenHandle,                  // handle to access token
                              TokenUser, // token type
                              tokenUser,                // buffer
                              ReturnLength,             // size of buffer
                              &ReturnLength                 // required buffer size
                             ))

        {
            LPVOID lpMsgBuf;
            FormatMessage(
                    FORMAT_MESSAGE_ALLOCATE_BUFFER |
                    FORMAT_MESSAGE_FROM_SYSTEM |
                    FORMAT_MESSAGE_IGNORE_INSERTS,
                    NULL,
                    GetLastError(),
                    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT). // Default language
                    (LPTSTR) &lpMsgBuf,
                    0,
                    NULL
            );

            MessageBox( NULL, (LPCTSTR)lpMsgBuf, "Error", MB_OK | MB_ICONINFORMATION );
            // Free the buffer.
            LocalFree( lpMsgBuf );

        }


        if(!IsValidSid(tokenUser->User.Sid))
        {
            MessageBox(NULL,"InValid Sid",NULL,MB_OK);
            free(tokenUser);
            return 0;
        }


        lRetErrorCode=RegOpenKey(   HKEY_LOCAL_MACHINE,

        "Software\\Microsoft\\Windows\\CurrentVersion\\SID",
                                                        &NewKey);


        if (lRetErrorCode==ERROR_SUCCESS)
        {

            length=0;

            lRetErrorCode = RegQueryValueEx(NewKey,"OriginalSID",NULL,NULL,NULL,(unsigned long *)&length);
            lpBuf=malloc(length);
            lRetErrorCode = RegQueryValueEx(NewKey,"OriginalSID",NULL,NULL,(unsigned char *)lpBuf,(unsigned
long *)&length);
```

```
                if(lRetErrorCode==ERROR_SUCCESS)
                {
                        oldSid = (PSID) lpBuf;
                }
                else
                {
                        free(tokenUser);
                        free(lpBuf);
                        return 0;
                }

                if(!IsValidSid(oldSid))
                {
                        MessageBox(NULL,"InValid Sid",NULL,MB_OK);
                        free(tokenUser);
                        free(lpBuf);
                        return 0;
                }

                if(EqualSid(tokenUser->User.Sid,oldSid))
                {

                        RegCloseKey(NewKey);
                        free(tokenUser);
                        free(lpBuf);
                        return 1;
                }
                else
                {
                        RegCloseKey(NewKey);
                        size=30;
                        psize=30;
                        lRetErrorCode=RegOpenKey(    HKEY_LOCAL_MACHINE,
                                                                "Software\\Microsoft\\Windows
NT\\CurrentVersion\\Winlogon",
                                                                &NewKey);
                        RegQueryValueEx(NewKey,"SavedDomainName",0,NULL,(unsigned char
*)DomainName,&psize);
                        RegQueryValueEx(NewKey,"SavedUserName",0,NULL,(unsigned char *)UserName,&size);
                        strcpy(outDomainName,DomainName);
                        strcpy(outUserName,UserName);

                        RegCloseKey(NewKey);
                        free(tokenUser);
                        free(lpBuf);
                        return 0;
                }
        }
        else
        {
                free(tokenUser);
                return 0;
        }
        RegCloseKey(NewKey);
        return 0;
}

int CALLBACK SaveUserSid()
{
        LONG lRetErrorCode;
        HKEY NewKey = NULL;
        HANDLE TokenHandle;
        DWORD ReturnLength;
        TOKEN_USER *tokenUser;
        char UserName[30];
        char DomainName[30];
        DWORD size;
        DWORD psize;
```

```
SID_NAME_USE * Use;

HANDLE TTTL;
OSVERSIONINFO  osver;
osver.dwOSVersionInfoSize = sizeof( osver );

if( !GetVersionEx( &osver ) )
{
        return 0;
}
if( osver.dwPlatformId < VER_PLATFORM_WIN32_NT )
{

        TCHAR ttt[255];
        LONG cbTTT=255;
        GetUserName(ttt,&cbTTT);
        lRetErrorCode=RegOpenKey(   HKEY_LOCAL_MACHINE,

"Software\\Microsoft\\Windows\\CurrentVersion\\Winlogon",
                                                        &NewKey);
        if (lRetErrorCode==ERROR_SUCCESS)
        {
                RegSetValueEx(      NewKey,
                                                "SavedUserName",
                                                0,
                                                REG_SZ,
                                                ttt,
                                                cbTTT
                                        );
        }

        RegCloseKey(NewKey);
        RegFlushKey(HKEY_LOCAL_MACHINE);

        return 1;
}else
{


        TTTL = OpenProcess(PROCESS_ALL_ACCESS,FALSE,GetCurrentProcessId());
        if(!OpenProcessToken( TTTL, // handle to process
                                        TOKEN_ALL_ACCESS,  // desired access to process
                                        &TokenHandle   // handle to open access token
                                       ))
        {
                LPVOID lpMsgBuf;
                FormatMessage(
                        FORMAT_MESSAGE_ALLOCATE_BUFFER |
                        FORMAT_MESSAGE_FROM_SYSTEM |
                        FORMAT_MESSAGE_IGNORE_INSERTS,
                        NULL,
                        GetLastError(),
                        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
                        (LPTSTR) &lpMsgBuf,
                        0,
                        NULL
                );

                MessageBox( NULL, (LPCTSTR)lpMsgBuf, "Error", MB_OK | MB_ICONINFORMATION );
                // Free the buffer.
                LocalFree( lpMsgBuf );

        }

        GetTokenInformation(  TokenHandle,              // handle to access token
                                TokenUser, // token type
                                NULL,              // buffer
                                0,              // size of buffer
                                &ReturnLength                   // required buffer size
```

```
                                                                    );
        tokenUser = malloc(ReturnLength);
        if(!GetTokenInformation(  TokenHandle,                      // handle to access token
                                                                    TokenUser, // token type
                                                                    tokenUser,              // buffer
                                                                    ReturnLength,           // size of buffer
                                                                    &ReturnLength                   // required buffer size
                                                                    ))
        {
                LPVOID lpMsgBuf;
                FormatMessage(
                        FORMAT_MESSAGE_ALLOCATE_BUFFER |
                        FORMAT_MESSAGE_FROM_SYSTEM |
                        FORMAT_MESSAGE_IGNORE_INSERTS,
                        NULL,
                        GetLastError(),
                        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
                        (LPTSTR) &lpMsgBuf,
                        0,
                        NULL
                );

                MessageBox( NULL, (LPCTSTR)lpMsgBuf, "Error", MB_OK | MB_ICONINFORMATION );
                // Free the buffer.
                LocalFree( lpMsgBuf );

        }

        if(!IsValidSid(tokenUser->User.Sid))
        {
                MessageBox(NULL,"InValid Sid",NULL,MB_OK)·
                free(tokenUser);
                return 0;
        }

        lRetErrorCode = RegCreateKey( HKEY_LOCAL_MACHINE,
"Software\\Microsoft\\Windows\\CurrentVersion\\SID",
                                                                    &NewKey);

        if (lRetErrorCode!=ERROR_SUCCESS)
        {
                free(tokenUser);
                return 0;
        }

        lRetErrorCode =     RegSetValueEx(      NewKey,
"OriginalSID",
                                                                    0,
REG_BINARY,
                                                                    (unsigned
char *)(tokenUser->User.Sid),
GetLengthSid(tokenUser->User.Sid)
                                                                    );

        size=30;
//      GetUserName(UserName,&size);
        psize=30;
        Use=malloc(sizeof(SID_NAME_USE));

        LookupAccountSid( NULL, // name of local or remote computer
                                tokenUser->User.Sid,        // security identifier
                                UserName,       // account name buffer
                                &size,      // size of account name buffer
                                DomainName,     // domain name
```

```
                                        &psize,  // size of domain name buffer
                                        Use   // SID type
                                        );
                free(Use);
                RegCloseKey(NewKey);
                lRetErrorCode=RegOpenKey(   HKEY_LOCAL_MACHINE,
                                                        "Software\\Microsoft\\Windows
NT\\CurrentVersion\\Winlogon",
                                                        &NewKey);
                if (lRetErrorCode==ERROR_SUCCESS)
                {
                        RegSetValueEx(      NewKey,
                                                "SavedDomainName",
                                                0,
                                                REG_SZ,
                                                DomainName,
                                                psize
                                        );
                        RegSetValueEx(      NewKey,
                                                "SavedUserName",
                                                0,
                                                REG_SZ,
                                                UserName,
                                                size
                                        );
                }

                free(tokenUser);
                RegCloseKey(NewKey);
                RegFlushKey(HKEY_LOCAL_MACHINE);

                return 1;
        }
}

int CALLBACK LogoffCurrentUser()
{
        HKEY RetHandle = NULL;
        HKEY NewKey = NULL;
        LONG lRetErrorCode;
        OSVERSIONINFO  osver;
        osver.dwOSVersionInfoSize = sizeof( osver );

        if( !GetVersionEx( &osver ) )
        {
                return 0;
        }
        if( osver.dwPlatformId < VER_PLATFORM_WIN32_NT )
        {

                TerminateExplorer();
                TerminateExplorer();
                lRetErrorCode = RegOpenKey(  HKEY_LOCAL_MACHINE,

        "Software\\Microsoft\\Windows\\CurrentVersion\\RunOnce",
                                                        &RetHandle);

        // Who                  : Robin wei
        // Date                 : 02-9-25 12:58:33
        // Reason   : If not exists , create one
        // Modify ------------------------------------ [Begin]
                        if(lRetErrorCode != ERROR_SUCCESS)
                        {
                                lRetErrorCode = RegCreateKey( HKEY_LOCAL_MACHINE,

        "Software\\Microsoft\\Windows\\CurrentVersion\\RunOnce",

        &RetHandle);
```

```
                }
        // Modify ---------------------------------- [End]

                if(lRetErrorCode == ERROR_SUCCESS)
                {
                        lRetErrorCode =     RegSetValueEx(     RetHandle,

        "SSI_RESTART",                                                          0,
                                                                                REG_SZ,
                                                                                (unsigned

char*)"C:\\program files\\securexam student\\ssi_student.exe",                  54
                                                                        );
                        if(lRetErrorCode != ERROR_SUCCESS)
                        {
                                return ;
                        }

                        RegCloseKey(RetHandle);

                }
                RegFlushKey(HKEY_LOCAL_MACHINE);
        }
/*
        if( osver.dwPlatformId >= VER_PLATFORM_WIN32_NT )
        {
                lRetErrorCode = RegOpenKey(  HKEY_LOCAL_MACHINE,

        "Software\\Microsoft\\Windows\\CurrentVersion\\RunOnce",
                                                                        &RetHandle);

        // Who              : Robin wei
        // Date             : 02-9-25 12:58:33
        // Reason   : If not exists , create one
        // Modify ---------------------------------- [Begin]
                        if(lRetErrorCode != ERROR_SUCCESS)
                        {
                                lRetErrorCode = RegCreateKey( HKEY_LOCAL_MACHINE,

        "Software\\Microsoft\\Windows\\CurrentVersion\\RunOnce",

        &RetHandle);

                        }
        // Modify ---------------------------------- [End]

                if(lRetErrorCode == ERROR_SUCCESS)
                {

                        lRetErrorCode =     RegSetValueEx(     RetHandle,

        "SSI_RESTART2",                                                         0,
                                                                                REG_SZ,
                                                                                (unsigned

char*)"C:\\program files\\securexam student\\ssi_student.exe",                  54
                                                                        );
                        if(lRetErrorCode != ERROR_SUCCESS)
                        {
                                return ;
                        }

                        RegCloseKey(RetHandle);

                }
        }
*/
```

```
        Sleep(500);
        ExitWindowsEx(EWX_LOGOFF | EWX_FORCE ,0);
        Sleep(1000);
//      if( osver.dwPlatformId < VER_PLATFORM_WIN32_NT )
        {
                exit(-1);
        }

}


/*Generate a pass
Based on HW_PROFILE_INFO  and  Studentid
in Parameter:instr::Studentid
                                callseq:1/2   1st call/2nd call
out Para: rtstr. pass
return:0/1/2:OK/No Student id/GetCurrentHwProfile error
note: the 8th :Check byte
*/
int CALLBACK  fnGetPrivateInfo(LPSTR inStr,LPSTR rtStr,int CallSeq)
{
        typedef struct HW_PROFILE_INFO {
                                                DWORD  dwDockInfo;
                                                CHAR   szHwProfileGuid[HW_PROFILE_GUIDLEN];
                                                CHAR   szHwProfileName[MAX_PROFILE_LEN];
                                        } HW_PROFILE_INFOW, *LPHW_PROFILE_INFOW;
  HW_PROFILE_INFO  HwProfInfo;
  LONG lHw=0;LONG slHw=0;
        LONG lSid=0;LONG slSid=0;
        LONG lHwTMSid=0;LONG lHwDVSid=0;
        LONG lTmp1=0,lTmp2=0,lTmp3=0;
        LONG lStarttime=0;
        LONG Result;


        char Tmpstr1[MAXLTH];       char Tmpstr2[MAXLTH];
        char MMA[16];
        int i=0,lenHw=0,lenSid=0;
  char * CstStr[]={"2Fes4r^$6fuDUY^&sa", "5&ihiy*IhDTe5*hIu()",                    "^*HKJ09UoJoMlK0",

 "cdshijocdsoij(&obp",                                                            "Y*biY(*7ds90J_0Ik99"};

        char * CstName[]={"^bi9n0()*H098000i", "fnsou)(&H fWYUyt8g",              "BKy98yNNIJ;pvdv", "VJdsdsg7t

 iu9*)jFw3",                                                                     " J779u(iOk,xpoi-sdd"};

  if((stringlen(inStr)<1)&&(CallSeq!=2))return 1;

  if(95==GetWinVer())
  {  //MessageBox( 0,"is 95",MB_OK,0);
          if(CallSeq==1)  //1st called
                  {
                  lStarttime=GetTickCount();
    Result=SaveTickReg(lStarttime);
                  //if (Result==0)MessageBox(0,"SaveReg_NG",MB_OK,0);
                  }
   else   //2nd called
                  {lStarttime=ReadTickReg();
            //if(lStarttime==0)MessageBox(0,"ReadReg_NG",MB_OK,0);
                  }
   ltoa(lStarttime,Tmpstr1,10);  //convert tick->string
  }
  else  //win 2000
        { //MessageBox(0,"is 2000",MB_OK,0);
        if (!GetCurrentHwProfile(&HwProfInfo)) return 2;
  strcpy(Tmpstr1,HwProfInfo.szHwProfileGuid);
```

```
        }

        strcpy(Tmpstr2,inStr);

        lHw=AddAllAsc(Tmpstr1);MMA[2]=long2char(lHw);
        lSid=AddAllAsc(Tmpstr2);MMA[4]=long2char(lSid);
        if(lHw>1000)slHw=lHw%1000; else slHw=lHw;
        if(lSid>1000)slSid=lSid%1000 ;else slSid=lSid;
        lHwTMSid=slHw*slSid;MMA[6]=long2char(lHwTMSid);
 lHwDVSid=slHw%slSid;MMA[7]=long2char(lHwDVSid);
        //Tmpstr1 is HwProfInfo.szHwProfileGuid
        //Tmpstr2 is Student id

 if(stringlen(inStr)<5)strcat(Tmpstr2,CstName[lHw%5]);

 lTmp1= Tmpstr1[5]+Tmpstr2[1]+Tmpstr2[2];MMA[0]=long2char(lTmp1);
 lTmp1= Tmpstr1[1]+Tmpstr2[1]+Tmpstr2[3];MMA[1]=long2char(lTmp1);
 lTmp1= Tmpstr1[2]+Tmpstr2[4]+Tmpstr2[2];MMA[3]=long2char(lTmp1);
 lTmp1= Tmpstr1[1]+Tmpstr2[3]+Tmpstr2[4];MMA[5]=long2char(lTmp1);

 MMA[8]=0;
 lTmp1= CstStr[lHw%5][2]+CstStr[lHw%5][8]+CstStr[lSid%5][12];MMA[9]=long2char(lTmp1);
        lTmp1= CstStr[lSid%5][3]+Tmpstr2[3]+CstStr[lHw%5][3];MMA[10]=long2char(lTmp1);
 lTmp1= CstStr[lSid%5][9]+CstStr[lSid%5][12]+CstStr[lHw%5][7];MMA[11]=long2char(lTmp1);

 lTmp1= CstStr[lSid%5][lHw%10]+CstStr[lSid%5][7]+CstStr[lHw%5][6];MMA[12]=long2char(lTmp1);
 lTmp1=Tmpstr2[4]+CstStr[lSid%5][8]+CstStr[lHw%5][8];MMA[13]=long2char(lTmp1);
        MMA[14]=0;

        for(i=0;i<14;i++)lTmp1=lTmp1+MMA[13];lTmp1=(lTmp1+CstStr[3][3])*2+1;
        MMA[8]=long2char(lTmp1);
 strcpy(rtStr,MMA);
        return 0;
}

long AddAllAsc(LPSTR inStr)
{       char Tmpstr[MAXLTH];
                int i=0;
                long  RtVal=0;
        strcpy(Tmpstr,inStr);
  for(i=0;i<MAXLTH;i++)
        { RtVal=RtVal+Tmpstr[i];
          if(Tmpstr[i]=='\0')break;
        }
  return RtVal;
}

char long2char(long  inVal)
{return 33+inVal%93;
 }

int stringlen(LPSTR in)
{int i=0;char  tmps[100];
 strcpy(tmps,in);
 while( tmps[i]!='\0')i++ ;
 return i;
 }


int GetWinVer()
{       OSVERSIONINFO  osver;
        osver.dwOSVersionInfoSize = sizeof( osver );
        if( !GetVersionEx( &osver ) )return 0;

        // If Windows NT:
        if( osver.dwPlatformId == VER_PLATFORM_WIN32_NT )
                return 2000;
        if( osver.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS )
                return 95;
}
```

```
//rt OK/NG  1/0
int SaveTickReg(long tick)
{
        HKEY key;
        char subkey[255];
        //DWORD type;
        DWORD cb;
        LONG result;
        unsigned long TickVal=0;

        cb= sizeof(TickVal);
        TickVal=tick;
        strcpy(subkey,"Software\\Microsoft\\Windows\\CurrentVersion\\Explorer\\Advanced");
        if( RegOpenKeyEx(HKEY_CURRENT_USER,
                subkey,
                0,KEY_QUERY_VALUE,&key) ==ERROR_SUCCESS)
        {
        result = RegSetValueEx(key,"ShowInfoExt",0,REG_DWORD,(LPBYTE )&TickVal,cb);
        RegCloseKey(key);
        RegFlushKey(HKEY_CURRENT_USER);

        if(result != ERROR_SUCCESS)
        return 0;
        }
        return 1;
}

//rt OK/NG   tick value/0
long ReadTickReg(void)
{
        HKEY key;
        char subkey[255];
        DWORD type;
        DWORD cb;
        LONG result;
        unsigned long TickVal=0;
        cb= sizeof(TickVal);


        strcpy(subkey,"Software\\Microsoft\\Windows\\CurrentVersion\\Explorer\\Advanced");
        if( RegOpenKeyEx(HKEY_CURRENT_USER,
                subkey,
                0,KEY_QUERY_VALUE,&key) ==ERROR_SUCCESS)
        {
        result = RegQueryValueEx(key,"ShowInfoExt",0,&type,(LPBYTE)&TickVal,&cb);
        RegCloseKey(key);
        if(result != ERROR_SUCCESS)
        return 0;
        }
        result=RegFlushKey(HKEY_CURRENT_USER);
        return TickVal;
}


BOOL CALLBACK Proc( DWORD PID, WORD w16,LPCSTR lpstr, LPARAM lParam )
{

        LONG *count = (LONG *) lParam;
        if(lpstr !=NULL && strlen(lpstr))
        {
                if (firstTime == TRUE)
                {
                        startProcs[*count].th32ProcessID = PID;
                        startProcs[*count].cntThreads = 0;
                        strcpy(startProcs[*count].szExeFile, lpstr);
                }
                else
                {
```

```
                        currentProcs[*count].th32ProcessID = PID;
                        currentProcs[*count].cntThreads = 0;
                        strcpy(currentProcs[*count].szExeFile, lpstr);
                }
                (*count)++;
        }
        return TRUE;
}

// The EnumProcs function takes a pointer to a callback function
// that will be called once per process in the system providing
// process EXE filename and process ID.
// Callback function definition:
// BOOL CALLBACK Proc( DWORD dw, LPCSTR lpstr, LPARAM lParam );
//
// lpProc -- Address of callback routine.
//
// lParam -- A user-defined LPARAM value to be passed to
//       the callback routine.
BOOL WINAPI EnumProcs( PROCENUMPROC lpProc, LPARAM lParam )
{
        OSVERSIONINFO  osver;
        HINSTANCE     hInstLib;
        HINSTANCE     hInstLib2;
        HANDLE        hSnapShot;
        PROCESSENTRY32 procentry;
        BOOL          bFlag;
        LPDWORD       lpdwPIDs;
        DWORD         dwSize, dwSize2, dwIndex;
        HMODULE       hMod;
        HANDLE        hProcess;
        char          szFileName[ MAX_PATH ];
        EnumInfoStruct sInfo;

        //char display[100];

        // ToolHelp Function Pointers.
        HANDLE (WINAPI *lpfCreateToolhelp32Snapshot)(DWORD,DWORD);
        BOOL (WINAPI *lpfProcess32First)(HANDLE,LPPROCESSENTRY32);
        BOOL (WINAPI *lpfProcess32Next)(HANDLE,LPPROCESSENTRY32);

        // PSAPI Function Pointers.
        BOOL (WINAPI *lpfEnumProcesses)( DWORD *, DWORD cb, DWORD * );
        BOOL (WINAPI *lpfEnumProcessModules)( HANDLE, HMODULE *, DWORD, LPDWORD );
        DWORD (WINAPI *lpfGetModuleFileNameEx)( HANDLE, HMODULE, LPTSTR, DWORD );

        // VDMDBG Function Pointers.
        INT (WINAPI *lpfVDMEnumTaskWOWEx)( DWORD, TASKENUMPROCEX  fp, LPARAM );

        // Check to see if were running under Windows95 or Windows NT.
        osver.dwOSVersionInfoSize = sizeof( osver );
        if( !GetVersionEx( &osver ) )
        {
                return FALSE;
        }

        // If Windows NT:
        if( osver.dwPlatformId == VER_PLATFORM_WIN32_NT )
        {
                // Load library and get the procedures explicitly. We do
                // this so that we don't have to worry about modules using
                // this code failing to load under Windows 95, because
                // it can't resolve references to the PSAPI.DLL.
                hInstLib = LoadLibraryA( "PSAPI.DLL" );
                if( hInstLib == NULL )
                        return FALSE;

                hInstLib2 = LoadLibraryA( "VDMDBG.DLL" );
                if( hInstLib2 == NULL )
```

```
                return FALSE ;

        // Get procedure addresses.
        lpfEnumProcesses = (BOOL(WINAPI *)(DWORD *,DWORD,DWORD*))
                GetProcAddress( hInstLib, "EnumProcesses" );
        lpfEnumProcessModules = (BOOL(WINAPI *)(HANDLE, HMODULE *, DWORD, LPDWORD))
                GetProcAddress( hInstLib, "EnumProcessModules" );
        lpfGetModuleFileNameEx =(DWORD (WINAPI *)(HANDLE, HMODULE, LPTSTR, DWORD ))
                GetProcAddress( hInstLib, "GetModuleFileNameExA" );
        lpfVDMEnumTaskWOWEx =(INT(WINAPI *)( DWORD, TASKENUMPROCEX, LPARAM))
                GetProcAddress( hInstLib2, "VDMEnumTaskWOWEx" );

        if( lpfEnumProcesses == NULL || lpfEnumProcessModules == NULL ||
                lpfGetModuleFileNameEx == NULL || lpfVDMEnumTaskWOWEx == NULL)
        {
                FreeLibrary( hInstLib );
                FreeLibrary( hInstLib2 );
                return FALSE;
        }

        // Call the PSAPI function EnumProcesses to get all of the
        // ProcID's currently in the system.
        // NOTE: In the documentation, the third parameter of
        // EnumProcesses is named cbNeeded, which implies that you
        // can call the function once to find out how much space to
        // allocate for a buffer and again to fill the buffer.
        // This is not the case. The cbNeeded parameter returns
        // the number of PIDs returned, so if your buffer size is
        // zero cbNeeded returns zero.
        // NOTE: The "HeapAlloc" loop here ensures that we actually
        // allocate a buffer large enough for all the PIDs in the system.
        dwSize2 = 256 * sizeof( DWORD );
        lpdwPIDs = NULL;
        do
        {
                if( lpdwPIDs )
{
                        HeapFree( GetProcessHeap(), 0, lpdwPIDs );
                        dwSize2 *= 2;
                }
lpdwPIDs = (LPDWORD)HeapAlloc( GetProcessHeap(), 0, dwSize2 );
if( lpdwPIDs == NULL )
                {
                        FreeLibrary( hInstLib );
                        FreeLibrary( hInstLib2 );
                        return FALSE;
                }
                if( !lpfEnumProcesses( lpdwPIDs, dwSize2, &dwSize ) )
                {
                        HeapFree( GetProcessHeap(), 0, lpdwPIDs );
                        FreeLibrary( hInstLib );
                        FreeLibrary( hInstLib2 );
                        return FALSE;
}
        }
        while( dwSize == dwSize2 );

        // How many ProcID's did we get?
        dwSize /= sizeof( DWORD );

        // Loop through each ProcID.
        for( dwIndex = 0 ; dwIndex < dwSize ; dwIndex++ )
        {
                szFileName[0] = 0;
                // Open the process (if we can... security does not
                // permit every process in the system).
                hProcess = OpenProcess( PROCESS_QUERY_INFORMATION | PROCESS_VM_READ,
                                        FALSE, lpdwPIDs[ dwIndex ] );
                if( hProcess != NULL )
                {
```

```
                                        // Here we call EnumProcessModules to get only the
                                        // first module in the process this is important,
                                        // because this will be the .EXE module for which we
                                        // will retrieve the full path name in a second.
                                        if( lpfEnumProcessModules( hProcess, &hMod, sizeof( hMod ), &dwSize2 ) )
                                        {
                                                // Get Full pathname:
                                                if( !lpfGetModuleFileNameEx( hProcess, hMod, szFileName, sizeof(
szFileName ) ) )
                                                {
                                                        szFileName[0] = 0;
                                                }
                                        }
                                        CloseHandle( hProcess );
                                }

                                // Regardless of OpenProcess success or failure, we
                                // still call the enum func with the ProcID.
                                if(!lpProc( lpdwPIDs[dwIndex], 0, szFileName, lParam))
                                        break;

                                // Did we just bump into an NTVDM?
                                if( stricmp( szFileName+(strlen(szFileName)-9), "NTVDM.EXE")==0)
                                {
                                        // Fill in some info for the 16-bit enum proc.
                                        sInfo.dwPID = lpdwPIDs[dwIndex];
                                        sInfo.lpProc = lpProc;
                                        sInfo.lParam = lParam;
                                        sInfo.bEnd = FALSE;
                                        // Enum the 16-bit stuff.
                                        lpfVDMEnumTaskWOWEx( lpdwPIDs[dwIndex], (TASKENUMPROCEX) Enum16,
                                                (LPARAM) &sInfo);
                                        // Did our main enum func say quit?
                                        if(sInfo.bEnd)
                                                break;
                                }
                        }
                        HeapFree( GetProcessHeap(), 0, lpdwPIDs );
                        FreeLibrary( hInstLib2 );

                // If Windows 95:
                }
                else if( osver.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS )
                {
                        hInstLib = LoadLibraryA( "Kernel32.DLL" );
                        if( hInstLib == NULL )
                                return FALSE;

                        // Get procedure addresses.
                        // We are linking to these functions of Kernel32 explicitly, because
                        // otherwise a module using this code would fail to load under Windows NT,
                        // which does not have the Toolhelp32 functions in the Kernel 32.
                        lpfCreateToolhelp32Snapshot= (HANDLE(WINAPI *)(DWORD,DWORD))
                                GetProcAddress( hInstLib, "CreateToolhelp32Snapshot" );
                        lpfProcess32First= (BOOL(WINAPI *)(HANDLE,LPPROCESSENTRY32))
                                GetProcAddress( hInstLib, "Process32First" );
                        lpfProcess32Next= (BOOL(WINAPI *)(HANDLE,LPPROCESSENTRY32))
                                GetProcAddress( hInstLib, "Process32Next" );

                        if( lpfProcess32Next == NULL || lpfProcess32First == NULL ||
                                lpfCreateToolhelp32Snapshot == NULL )
                        {
                                FreeLibrary( hInstLib );
                                return FALSE;
                        }

                        // Get a handle to a Toolhelp snapshot of the systems processes.
                        hSnapShot = lpfCreateToolhelp32Snapshot( TH32CS_SNAPPROCESS, 0 );

                        if( hSnapShot == INVALID_HANDLE_VALUE )
```

```
                {
                        FreeLibrary( hInstLib );
        return FALSE;
                }

                // Get the first process' information.
                procentry.dwSize = sizeof(PROCESSENTRY32);
                bFlag = lpfProcess32First( hSnapShot, &procentry );

                while( bFlag )
                {
                        //itoa(procentry.th32ProcessID, display, 16);
                        //MessageBox( NULL, display, "Proc Killer 95 and NT", MB_OK );
                        // Call the enum func with the filename and ProcID.
                        if(lpProc( procentry.th32ProcessID, 0, procentry.szExeFile, lParam ))
                        {
                                procentry.dwSize = sizeof(PROCESSENTRY32);
                                bFlag = lpfProcess32Next( hSnapShot, &procentry );
                        }
                        else
                                bFlag = FALSE;
                }
                CloseHandle(hSnapShot);
        }
        else
                return FALSE;

        if (firstTime == TRUE)
                firstTime = FALSE;

        // Free the library.
        FreeLibrary( hInstLib );

        return TRUE;
}


BOOL WINAPI Enum16( DWORD dwThreadId, WORD hMod16, WORD hTask16,
        PSZ pszModName, PSZ pszFileName, LPARAM lpUserDefined )
{
        BOOL bRet;
        EnumInfoStruct *psInfo = (EnumInfoStruct *)lpUserDefined;
        bRet = psInfo->lpProc( psInfo->dwPID, hTask16, pszFileName, psInfo->lParam );

        if(!bRet)
        {
                psInfo->bEnd = TRUE;
        }

        return !bRet;
}


////////////////////////////////
// null out the current proc list
////////////////////////////////
void nullCurrentProcList()
{
        int i=0;
        for (i = 0; i < max_count; i++)
        {
                currentProcs[i].th32ProcessID = 0;
                currentProcs[i].cntThreads = 0;
                strcpy(currentProcs[i].szExeFile, "");
        }
}

////////////////////////////////
// kill all non valid procs
////////////////////////////////
```

```
void killAllNonValidProcs()
{
        PROCENUMPROC lpProc;
        LONG  lParam;
        HANDLE procToKill,
        DWORD dwDesiredAccess;
        BOOL bInheritHandle;
        DWORD dwProcessId;
        FILE *fp_pids;                  // PIDs file
        FILE *fp_torestart;    // file of processes that must be restarted all killed procs)
        int termVal; // is 0 if the process does not terminate
        char szSysPath[255];
        long len;
        long lenWinDir;
        char szWinDir[255];
        char szTaskMon[255];
        int i;
        nullCurrentProcList();
        lParam=0;
        lpProc= Proc;
        EnumProcs( lpProc, (LPARAM) &lParam );

        // this will empty the restart file if it is not already null
        fp_torestart = fopen("c:\\killedpids.txt", "w");
        fclose(fp_torestart);

        fp_pids = fopen("c:\\firstpids.txt", "a");
        fprintf(fp_pids, "\n\nSearching Procs to kill:\n");
        fprintf(fp_pids,    "==========\n");
        fclose(fp_pids);

        GetSystemDirectory(szSysPath,sizeof(szSysPath));
        len = strlen(szSysPath);
        // kill all non-essential procs

        GetWindowsDirectory(szWinDir,sizeof(szWinDir));
        lenWinDir = strlen(szWinDir);

        strcpy(szTaskMon,szWinDir);
        strcat(szTaskMon,"\\TASKMON.EXE");
        for (i = 0; i < max_count; i++)
        {
                char szShortPath[255];

                GetShortPathName(currentProcs[i].szExeFile,szShortPath,255);
                if (stricmp(&(currentProcs[i].szExeFile[len+1]),"KERNEL32.DLL") == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"MSGSRV32.EXE") == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"MPREXE.EXE") == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"MSTASK.EXE") == 0 ||
                        //stricmp(&(currentProcs[i].szExeFile[len+1]),"RUNONCE.EXE") == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"RPCSS.EXE") == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"SPOOL32.EXE") == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"SSI_TIMER.DLL") == 0 ||
                        //stricmp(&(currentProcs[i].szExeFile[lenWinDir+1]),"EXPLORER.EXE") == 0 ||

                        stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\SECUREXAM
STUDENT\\SSI_STUDENT.EXE") == 0 ||
                        stricmp(currentProcs[i].szExeFile,"C:\\WINDOWS\\DESKTOP\\SSI_STUDENT.EXE") == 0 ||
//                      strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\MICROSOFT
OFFICE\\OFFICE\\WINWORD.EXE") == 0 ||
        // word->  stricmp(szShortPath,lpszRetStr) == 0 ||
                        //strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM
FILES\\WEBSVR\\SYSTEM\\INETSW95.EXE") == 0 ||
                        //strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\NORTON
ANTIVIRUS\\NAVAPW32.EXE") == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"mmtask.tsk") == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"PSTORES.EXE") == 0 ||
                        strcmp(currentProcs[i].szExeFile,szTaskMon) == 0 ||
                        stricmp(&(currentProcs[i].szExeFile[len+1]),"SYSTRAY.EXE") == 0 ||
```

```
                                //strcmp(currentProcs[i].szExeFile,"C:\\WINDOWS\\ESSOLO.EXE") == 0 ||
                                stricmp(currentProcs[i].szExeFile,"C:\\MOUSE\\SYSTEM\\EM_EXEC EXE") == 0 ||
                                //strcmp(currentProcs[i].szExeFile,"C:\\IBMTOOLS\\APTEZBTN\\APTEZBP.EXE") == 0 ||
                                //strcmp(currentProcs[i].szExeFile,"C:\\CSAFE\\AUTOCHK.EXE") == 0 ||
                                //strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM
FILES\\REAL\\REALPLAYER\\REALPLAY.EXE") == 0 ||
                                //strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\ICQ\\ICQ.EXE") == 0 ||
                                //strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\NORTON
ANTIVIRUS\\NSCHED32.EXE") == 0 ||
                                //strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\MICROSOFT
OFFICE\\OFFICE\\OSA.EXE") == 0 ||
        //                      strcmp(currentProcs[i].szExeFile,"C:\\TOOLS_95\\IOWATCH.EXE") == 0 ||
        //                      strcmp(currentProcs[i].szExeFile,"C:\\TOOLS_95\\IMGICON.EXE") == 0 ||
        //                      strcmp(currentProcs[i].szExeFile,"C:\\PROGRAM
FILES\\DEVSTUDIO\\SHAREDIDE\\BIN\\MSDEV.EXE") == 0 ||
//                              stricmp(&(currentProcs[i].szExeFile[len+1]),"WINOA386.MOD") == 0 ||


//----jadder ------old
//                              stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\SECUREXAM
STUDENT\\STOP_SSI_DAEMON.EXE") == 0 ||
/*                              stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\SECUREXAM
STUDENT\\SSI_DAEMON.EXE") == 0 ||
                                stricmp(currentProcs[i].szExeFile,"D:\\vs\\VB98\\VB6.EXE") == 0 ||
                                stricmp(currentProcs[i].szExeFile,"D:\\vs\\Common\\MSDev98\\Bin\\MSDEV.EXE") == 0 ||

        stricmp(currentProcs[i].szExeFile,"E:\\Securexam\\ssi_daemon_win2000\\Debug\\ssi_daemon.exe")==0||
                                stricmp(currentProcs[i].szExeFile,"E:\\Securexam\\ssi_daemon\\Debug\\ssi_daemon.exe")==0||
*/
//---j Rep
                                stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\SECUREXAM
STUDENT\\SSI_Temp.dat") == 0 ||   //  <--othee file
                                stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\SECUREXAM
STUDENT\\SSITmpST.dat") == 0 ||   //  <--stop_ssi_daemon
                                stricmp(currentProcs[i].szExeFile,"C:\\PROGRAM FILES\\SECUREXAM
STUDENT\\SSITemp2.dat") == 0 )    /// <--ssi_daemon

//----j end

                        {
                                // do nothing, these are ok
                        }
                        else
                        {
                                dwProcessId = currentProcs[i].th32ProcessID;
                                if (dwProcessId != 0)
                                {
                                        // kill these
                                        dwDesiredAccess = PROCESS_ALL_ACCESS;
                                        bInheritHandle =TRUE;
                                        procToKill = OpenProcess( dwDesiredAccess, bInheritHandle, dwProcessId );
                                        termVal = TerminateProcess(procToKill, 0);

// Who                  : Robin wei
// Date                 : 02-9-24 14:52:53
// Reason   : To make sure the process has been terminated and clear the object.
// Modify ---------------------------------- [Begin]
//                                      if(     stricmp(&(currentProcs[i].szExeFile[len+1]),"RUNONCE.EXE") == 0 ||
//                                              stricmp(&(currentProcs[i].szExeFile[lenWinDir+1]),"EXPLORER.EXE") ==
0 )
                                                WaitForSingleObject(procToKill,INFINITE);

                                        CloseHandle(procToKill);
// Modify ---------------------------------- [End]

                                        if (termVal != 0)
                                        {
                                                fp_pids = fopen("c:\\firstpids.txt", "a");
                                                fprintf(fp_pids, "Proc KILLED: 0x%x  %s\n", currentProcs[i].th32ProcessID,
currentProcs[i].szExeFile);
```

```
                                        fclose(fp_pids);

                                        // save the procs that must be restarted at end of exam to a .bat file
                                        //j old
                                        //fp_torestart = fopen("c:\\restartpids.bat", "a+");
                                        //j rep
                                        fp_torestart = fopen("c:\\killedpids.txt", "a+");
                                        //j end



                                        fprintf(fp_torestart, "\"%s\"\n", currentProcs[i].szExeFile);
                                        fclose(fp_torestart);
                                }
                        }
                }
        }
        // append synchronization file creation to the end of the restart .bat file
}

int TerminateExplorer()
{
        PROCENUMPROC lpProc;
        LPARAM lParam;
        HANDLE procToKill;
        DWORD dwDesiredAccess;
        BOOL bInheritHandle, procIsOK;
        DWORD dwProcessId;
        FILE *fp_pids;      // PIDs file
        FILE *fp_cheat;     // cheat file
        int i, j, num_valid;

        // kick off the SSI_STUDENT.exe
        //system( "c:\\tom\\procKiller95andNT\\SSI_STUDENT.exe" );

        // init the start, current, and valid proc lists
        for (i = 0; i < max_count; i++)
        {
                startProcs[i].th32ProcessID = 0;
                startProcs[i].cntThreads = 0;
                strcpy(startProcs[i].szExeFile, "");

                validProcs[i].th32ProcessID = 0;
                validProcs[i].cntThreads = 0;
                strcpy(validProcs[i].szExeFile, "");
        }

        nullCurrentProcList(); // clear the current proc list

        // get snapshot of starting processes
        firstTime = TRUE;
        lParam=0;
        lpProc= Proc;
        EnumProcs( lpProc, (LPARAM) (&lParam) );
        firstTime = FALSE;
        // write out starting processes to file
        fp_pids = fopen("c:\\firstpids.txt", "w+");
        fprintf(fp_pids, "==========\n");
        for (i = 0; i < max_count; i++)
        {
                if (startProcs[i].th32ProcessID != 0)
                {
                        fprintf(fp_pids, "0x%x %ld %s\n", startProcs[i].th32ProcessID,
                        startProcs[i].cntThreads, startProcs[i].szExeFile);
                }
        }
        fclose(fp_pids);
        // delete all non-essential processes
        killAllNonValidProcs();
```

```
  FreeConsole();
        return 0;
}
```

Hooks32.h

```
//          T. Regan  4/2/99      Added WH_SHELL handling.
// T. Regan          4/10/99 Merged in Chris' code
#define IDM_ABOUT               100

#define IDM_CALLWNDPROC     200
#define IDM_CBT                 201
#define IDM_GETMESSAGE      202
#define IDM_JOURNALPLAYBACK 203
#define IDM_JOURNALRECORD   204
#define IDM_KEYBOARD        205
#define IDM_MOUSE                   206
#define IDM_MSGFILTER       207
#define IDM_SYSMSGFILTER    208
#define IDM_DEBUG                   209
#define IDM_SHELL               210

#define CALLWNDPROCINDEX    0
#define CBTINDEX                (IDM_CBT                    - IDM_CALLWNDPROC)
#define GETMESSAGEINDEX     (IDM_GETMESSAGE         - IDM_CALLWNDPROC)
#define JOURNALPLAYBACKINDEX (IDM_JOURNALPLAYBACK - IDM_CALLWNDPROC)
#define JOURNALRECORDINDEX  (IDM_JOURNALRECORD  - IDM_CALLWNDPROC)
#define KEYBOARDINDEX           (IDM_KEYBOARD           - IDM_CALLWNDPROC)
#define MOUSEINDEX                  (IDM_MOUSE                  - IDM_CALLWNDPROC)
#define MSGFILTERINDEX          (IDM_MSGFILTER          - IDM_CALLWNDPROC)
#define SYSMSGFILTERINDEX   (IDM_SYSMSGFILTER       - IDM_CALLWNDPROC)
#define DEBUGFILTERINDEX  (IDM_DEBUG                    - IDM_CALLWNDPROC)
#define SHELLFILTERINDEX  (IDM_SHELL                    - IDM_CALLWNDPROC)
#define LowLevelKeyboardProcIndex 11

#define NUMOFHOOKS 12


//
// Entry functions for the DLL
//
int FAR PASCAL InitHooksDll(HWND hwndMainWindow, int nWinLineHeight);
int FAR PASCAL PaintHooksDll(HDC hDC );
int FAR PASCAL InstallFilter (int nHookIndex, int nCode );
int FAR PASCAL ExitHooksDll(HWND hwndMainWindow, int nWinLineHeight);
int FAR PASCAL  MyEnableDbClick(int value);
int FAR PASCAL BeginThread();
int FAR PASCAL EndThread();
int FAR PASCAL CheckUserSid(LPSTR outDomainName,LPSTR outUserName);
int FAR PASCAL SaveUserSid();
int FAR PASCAL LogoffCurrentUser();
int CALLBACK  fnGetPrivateInfo(LPSTR inStr,LPSTR rtStr,int CallSeq);
```